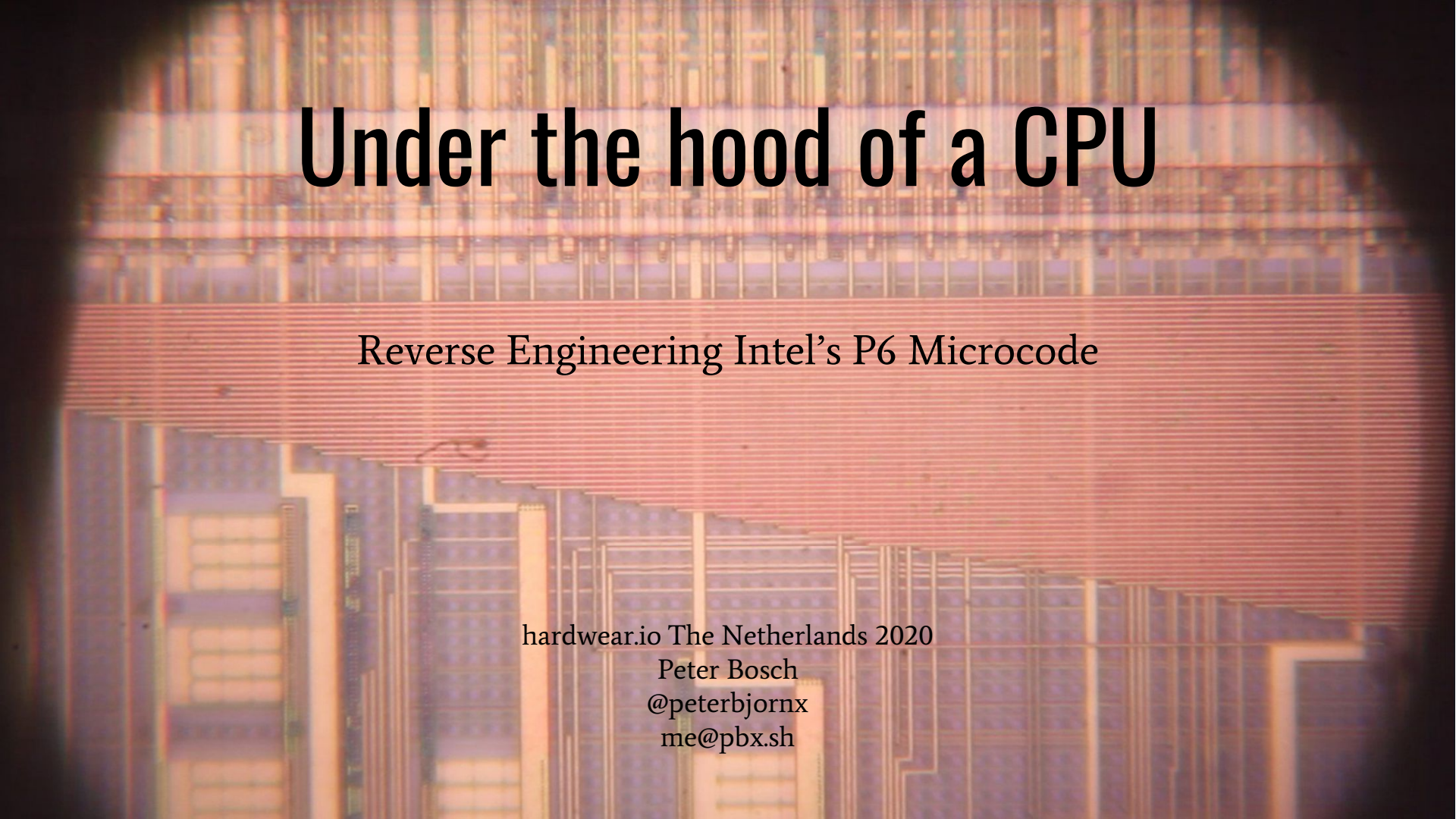


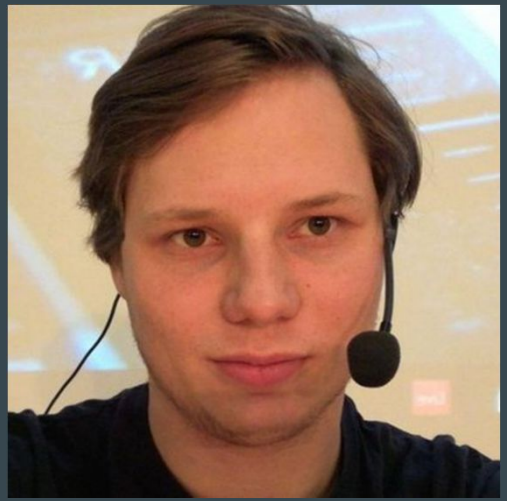
# Under the hood of a CPU

## Reverse Engineering Intel's P6 Microcode

hardwear.io The Netherlands 2020
Peter Bosch
@peterbjornx
me@pbx.sh

# About me

- Computer Science/Physics student at Leiden University
- Past work includes
  - Writing an emulator for the Intel ME
    - 36C3 Talk: Intel Management Engine Deep Dive
  - CVE-2019-11098 (Intel Boot Guard SPI bus TOCTOU vulnerability) (with @qrs)
    - HITB2019 Talk: Now You See It: TOCTOU Attacks Against Secure Boot and BootGuard



Twitter: @peterbjornx

E-mail: me@pbx.sh

# x86 != native instruction set

- MacroInstructions are converted to microinstructions (uops)
- Simple instructions map 1-to-1 : `ADD EAX, REG` becomes `EAX:= ADD EAX, REG`
- More complicated instructions yield multiple uops
- Even more complicated instructions and other architectural details invoke a full microprogram

# APPENDIX C
# PENTIUM® PRO PROCESSOR INSTRUCTION TO DECODER SPECIFICATION

| | | | |
|---|---|---|---|
| ST | comp ex | VERR m16 | comp ex |
| STOSB/W/D m8/16/32,m8/16/32 | 3 | VERR rm16 | comp ex |
| STR m16 | comp ex | VERW m16 | comp ex |
| STR rm16 | 4 | VERW rm16 | comp ex |
| SUB AL, mm8 | 1 | WB NVD | comp ex |
| SUB eAX, mm16/32 | 1 | WRMSR | comp ex |
| SUB m16/32, mm16/32 | 4 | XADD m16/32,r16/32 | comp ex |
| SUB m16/32,r16/32 | 4 | XADD m8,r8 | comp ex |
| AAS | 1 | ADD r16/32,rm16/32 | 1 |
| ADC AL, mm8 | 2 | ADD r8, mm8 | 1 |
| ADC eAX, mm16/32 | 2 | ADD r8,m8 | 2 |
| ADC m16/32, mm16/32 | 4 | ADD r8,rm8 | 1 |
| ADC m16/32,r16/32 | 4 | ADD rm16/32,r16/32 | 1 |
| ADC m8, mm8 | 4 | ADD rm8,r8 | 1 |

# Getting to these microprograms

- Download an update and extract it?

Each block is encoded differently, and the majority of the 2,000 bytes are not used as configuration program and SRAM micro-operation contents themselves are much smaller.[1] Final determination and validation of whether an update can be applied to a processor is performed during decryption via the processor.[16] Each microcode update is specific to a particular CPU revision, and is designed to be rejected by CPUs with a different stepping level. Microcode updates are encrypted to prevent tampering and to enable validation.[20]

With the Pentium there are two layers of encryption and the precise details explicitly *not* documented by Intel, instead being only known to less than ten employees.[21]

Each block is encoded differently, and the majority of the 2,000 bytes are not used as configuration program and SRAM micro-operation contents themselves are much smaller.[1] Final determination and validation of whether an update can be applied to a processor is performed during decryption via the processor.[16] Each microcode update is specific to a particular CPU revision, and is designed to be rejected by CPUs with a different stepping level. Microcode updates are encrypted to prevent tampering and to enable validation.[20]

With the Pentium there are two layers of encryption and the precise details explicitly *not* documented by Intel, instead being only known to less than ten employees.[21]

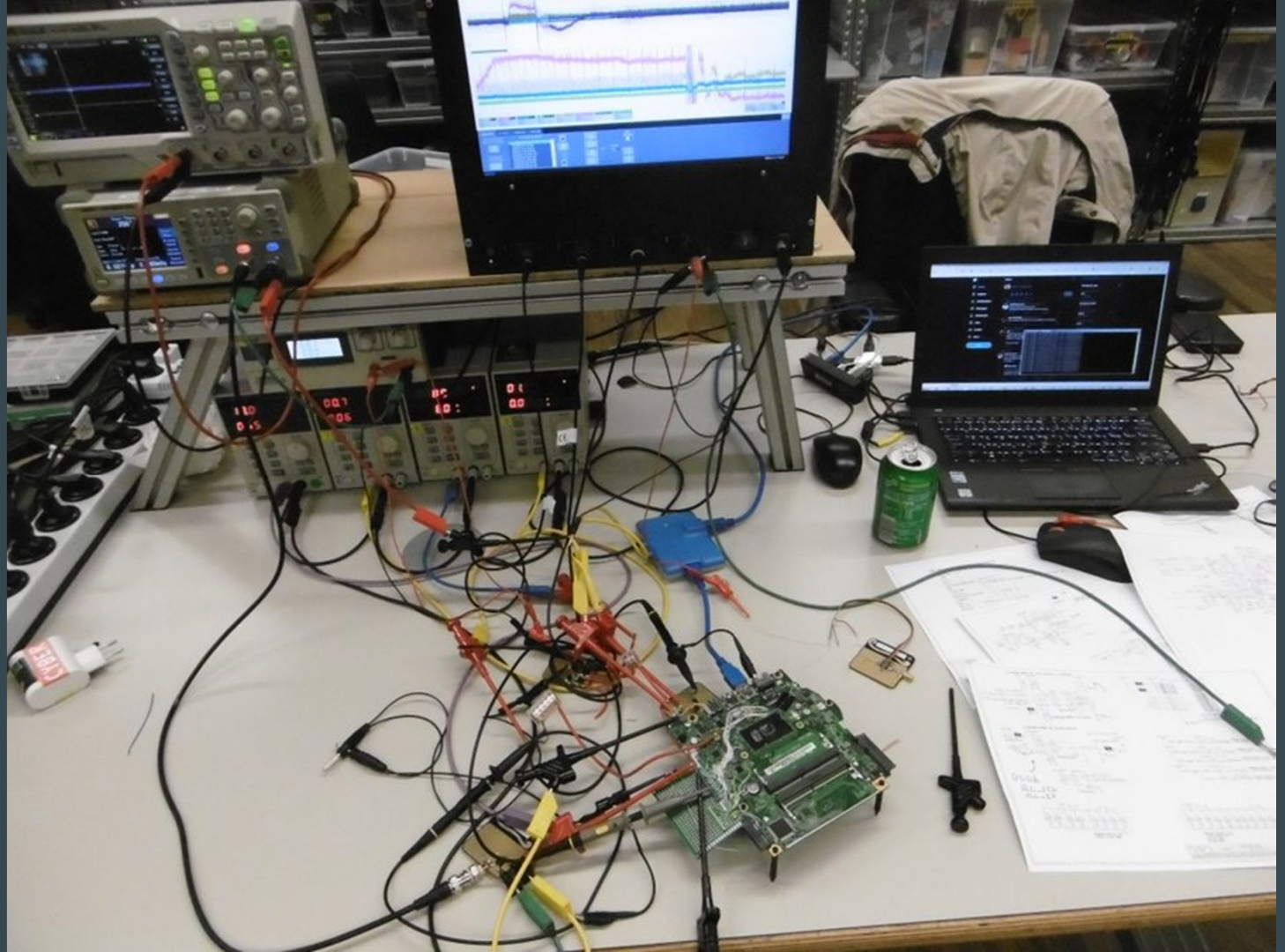
# Getting access to these microprograms

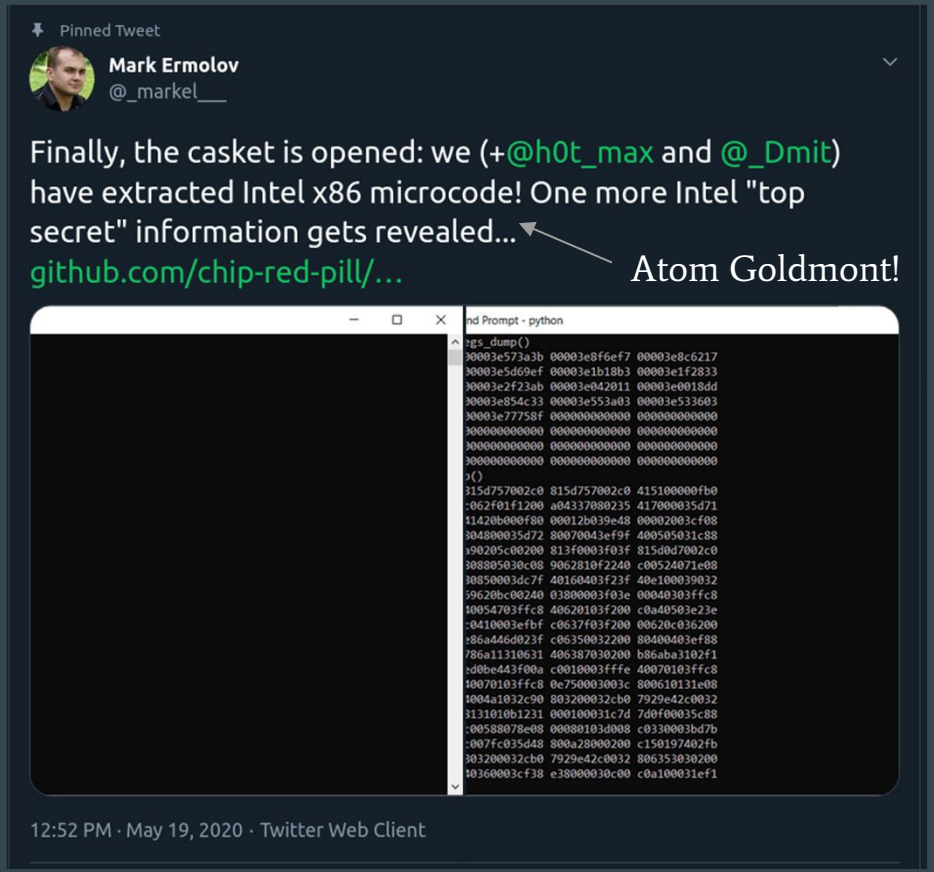
- ~~Download an update and extract it?~~
- Extract from running system?

# LDAT Ports

```
<_tdef deviceType="SNB" tables="StateDefs">
  <_tdefDevice steppings="A0,A1,B0,B2,C0,C1,D0,D1,D2,J0,J1,P0,Q0">
    <StateDefs>
      <StateDef>
        <State _name="ms_ram_by_rf" _description="The MS patch RAM">
          <Dimension _name="set" _instances="127:0">
            <Field _name="RF_9(31:0)" />
            <Field _name="RF_8(31:0)" />
            <Field _name="RF_7(31:0)" />
            <Field _name="RF_6(31:0)" />
            <Field _name="RF_5(31:0)" />
            <Field _name="RF_4(31:0)" />
            <Field _name="RF_3(31:0)" />
            <Field _name="RF_2(31:0)" />
            <Field _name="RF_1(31:0)" />
            <Field _name="RF_0(31:0)" />
          </Dimension>
        </State>
        <Operation Type="Read" Scope="set">
          <Procedure Name="Readms_ram_by_rf">
            <Param Name="set" />
            <LDatSetup UnitSel="0x3d3" />
```

More info: https://pbx.sh/ldat/






https://github.com/chip-red-pill/glm-ucode

# Getting access to these microprograms

- ~~Download an update and extract it?~~
- ~~Extract from running system?~~
- Extract from mask ROM?

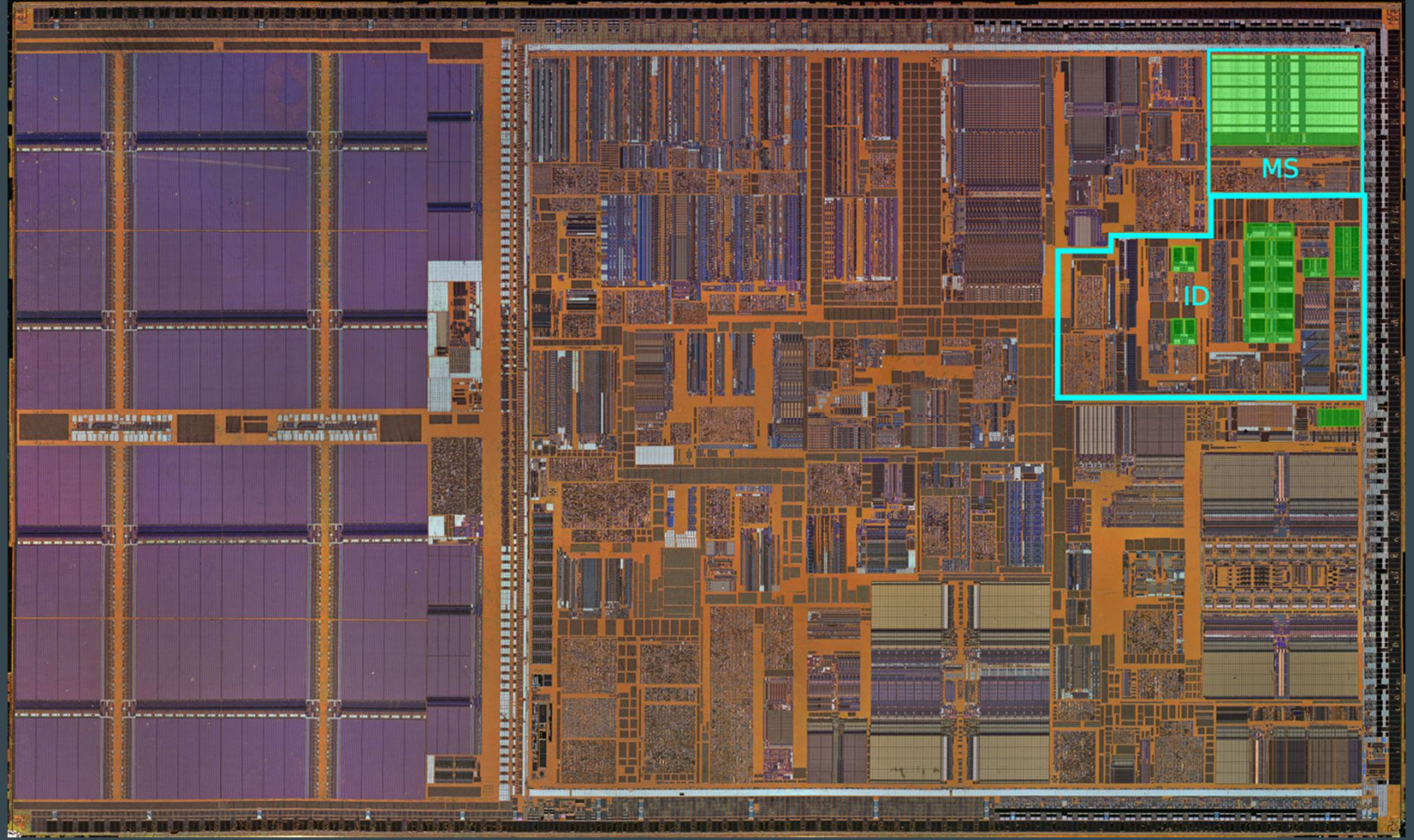
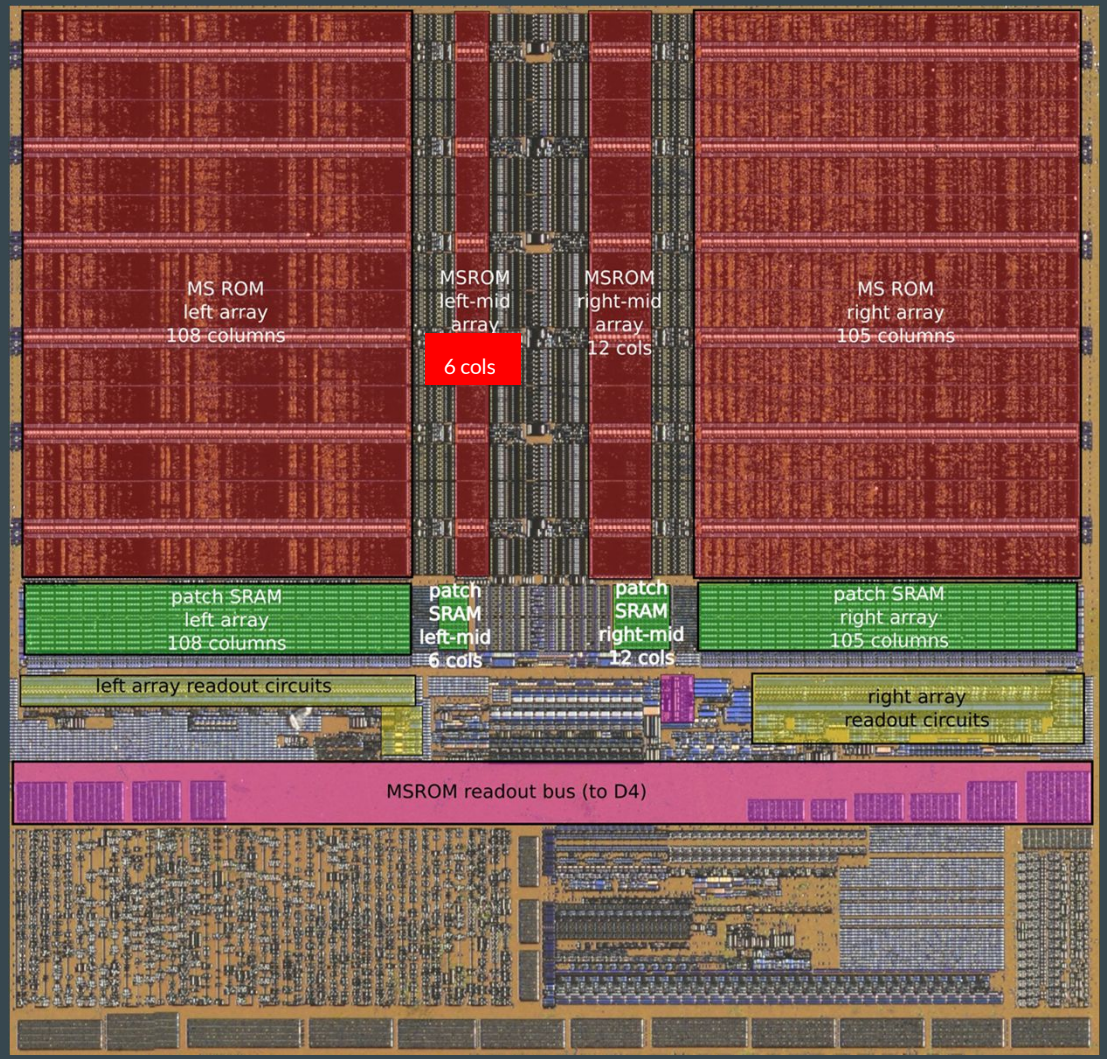
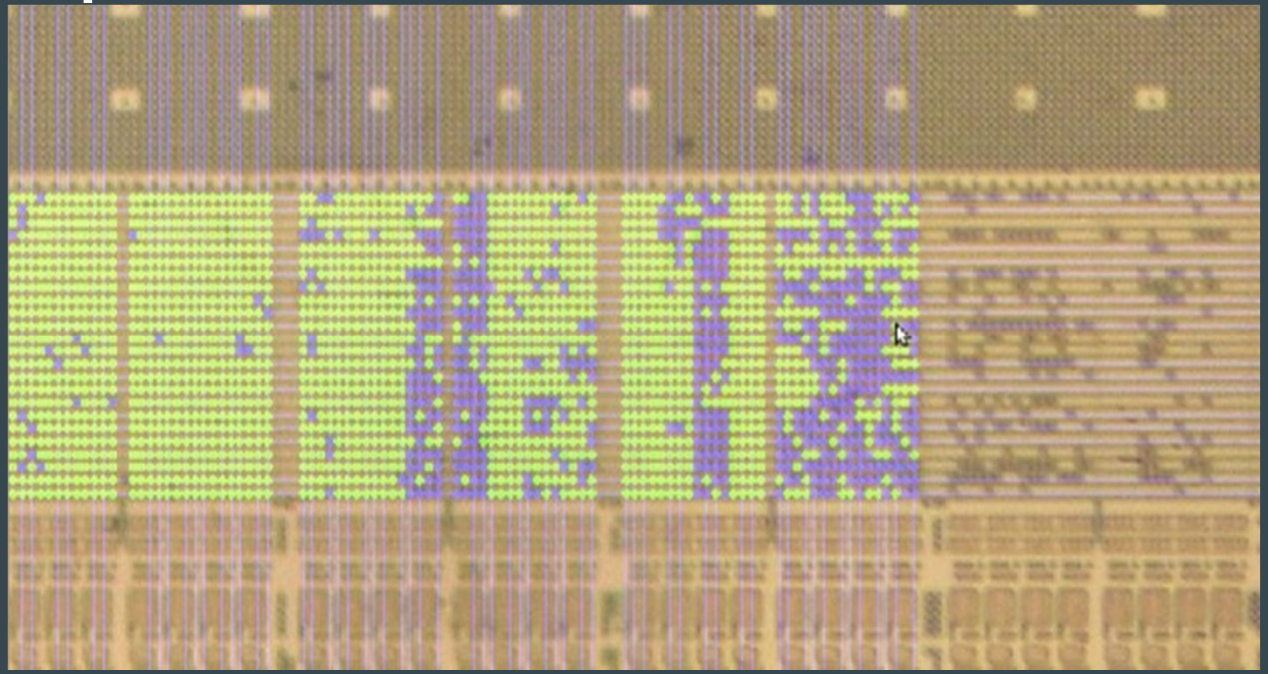



Image by Martijn Boer,
https://www.flickr.com/photos/sic66/42522440724/in/album-72157689303100124/

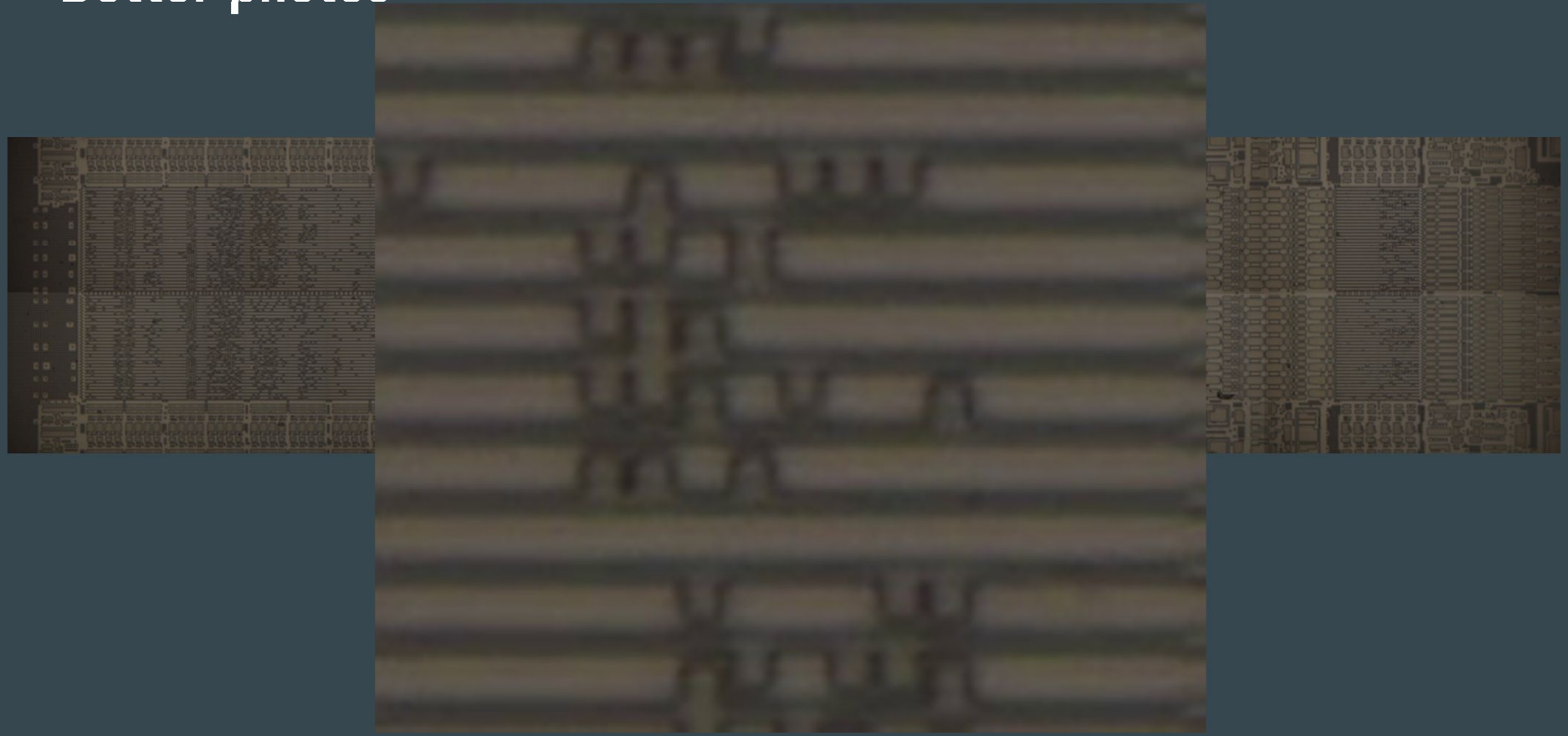

MS ROM
left array
108 columns
MSROM
left-mid
array
6 cols
MSROM
right-mid
array
12 cols
MS ROM
right array
105 columns
patch SRAM
left array
108 columns
patch
SRAM
left-mid
6 cols
patch
SRAM
right-mid
12 cols
patch SRAM
right array
105 columns
left array readout circuits
right array
readout circuits
MSROM readout bus (to D4)

# First attempt

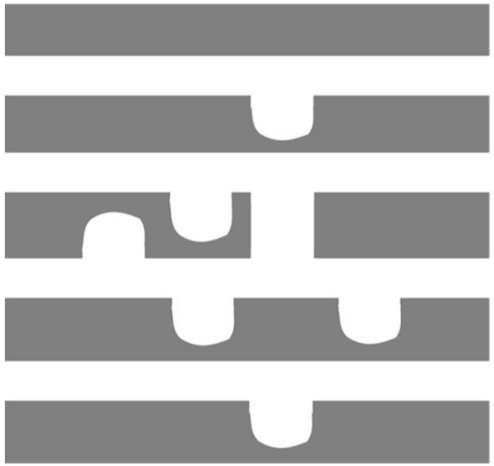


https://github.com/AdamLaurie/rompar

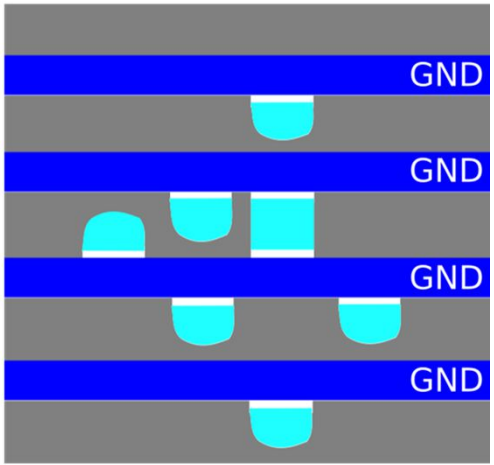
# Better photos



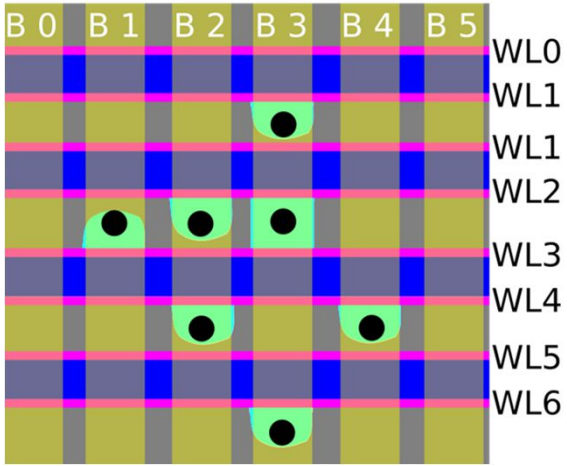
Trench oxide/Active area



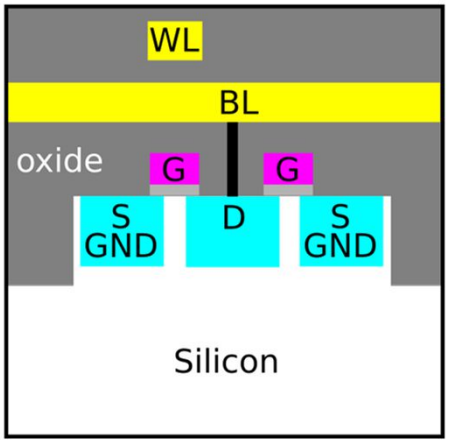
Implant (Source/Drain)

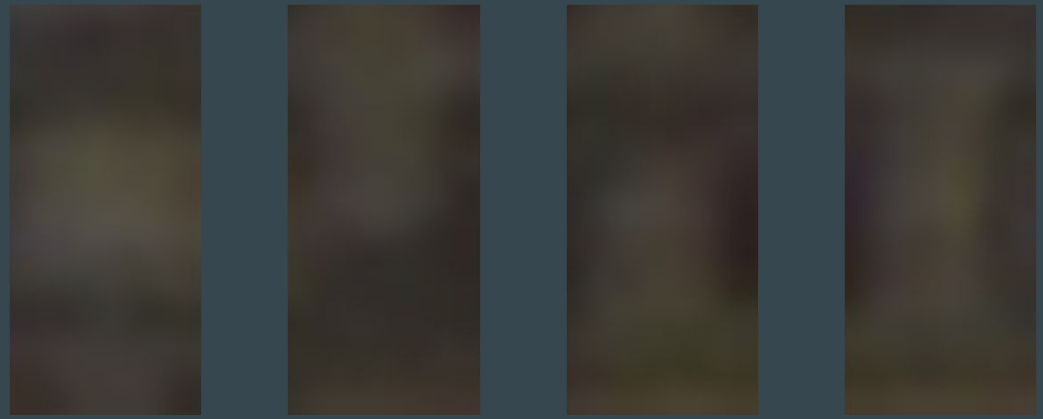



Metal 1, Poly (Gate)




Cross-section (Both bits set)





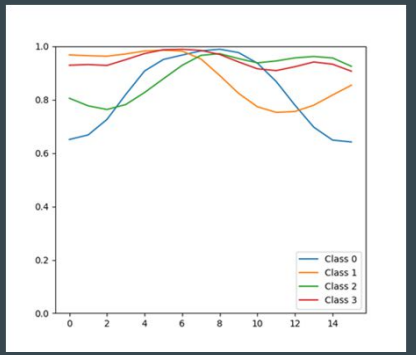
0 1 2 3

ROM image

Crop ROM cell pairs


Cell images

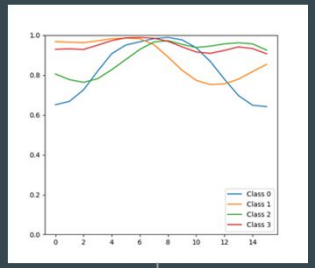
Average along Y axis and normalize

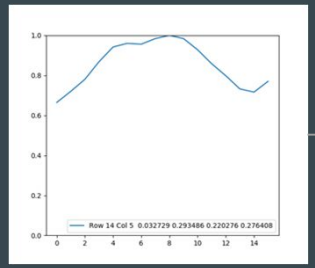



1D cell function

Reference cell functions

$r$

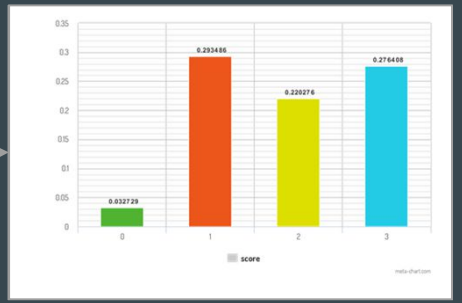


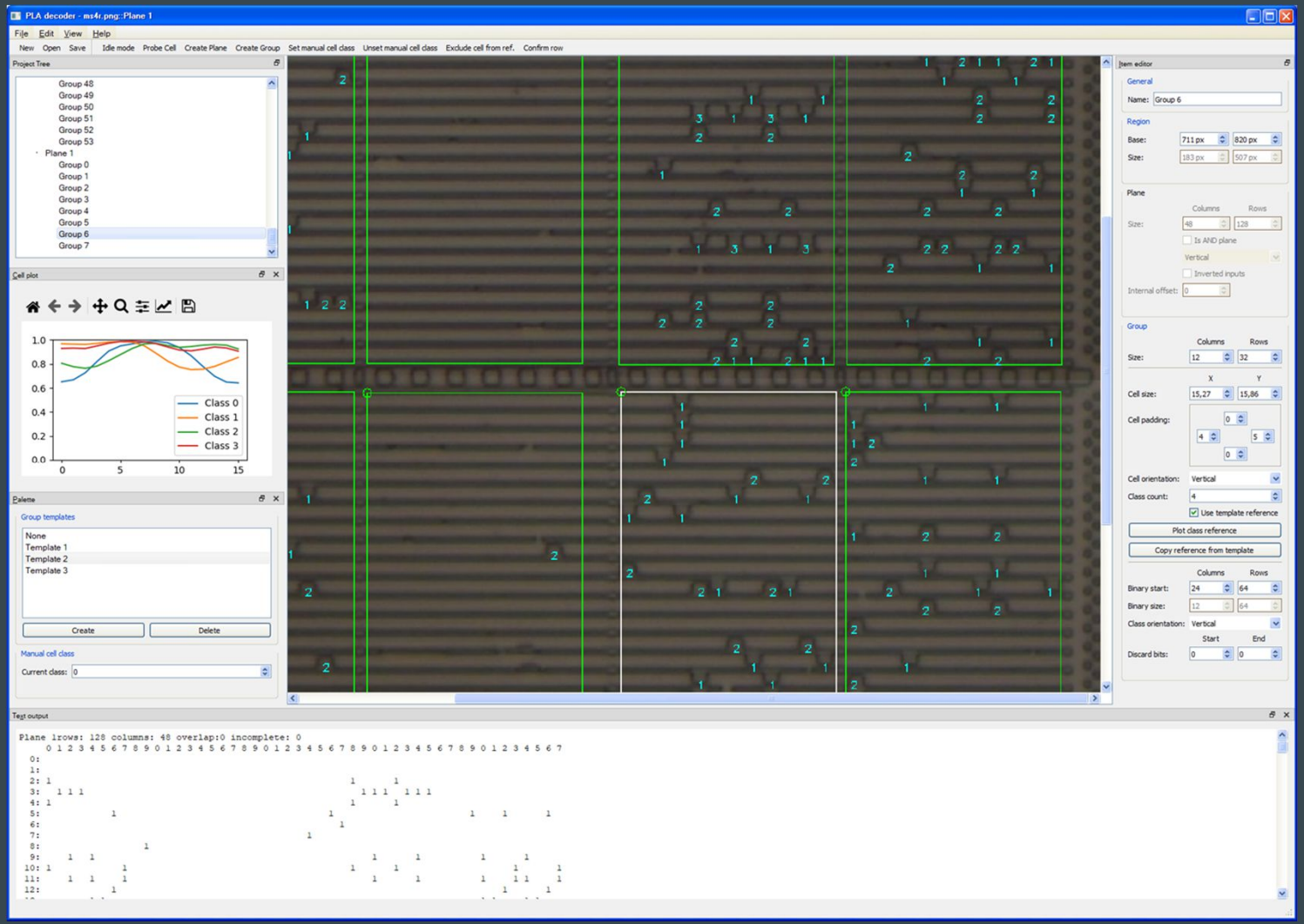

Unknown cell function

$x$

$$s_j = \sum_i \frac{(x_i - r_{i,j})^2}{r_{i,j}^2}$$



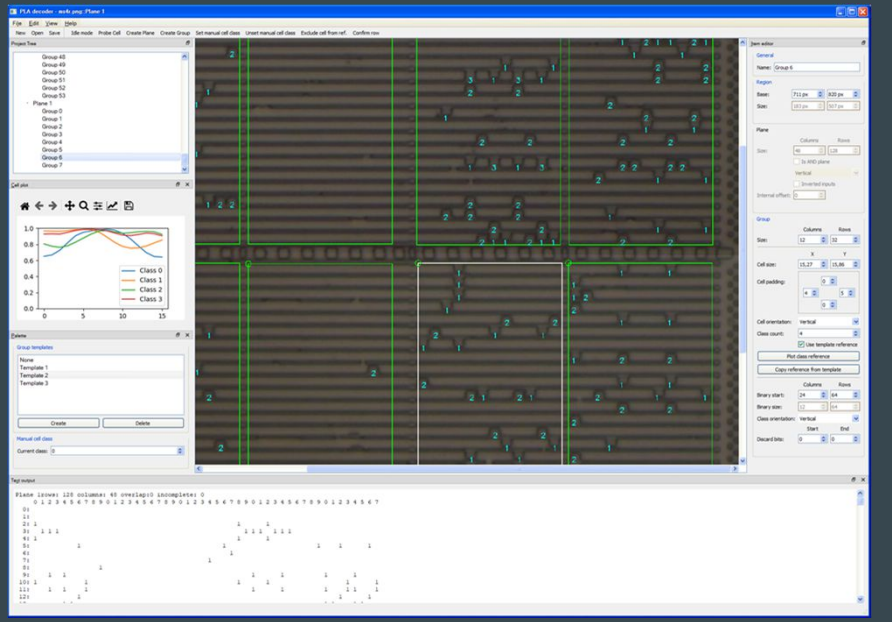

Cell scores


PLA decoder - ms4r.png::Plane 1
File Edit View Help
New Open Save Idle mode Probe Cell Create Plane Create Group Set manual cell class Unset manual cell class Exclude cell from ref. Confirm row
Project Tree
Group 48
Group 49
Group 50
Group 51
Group 52
Group 53
Plane 1
Group 0
Group 1
Group 2
Group 3
Group 4
Group 5
Group 6
Group 7
Cell plot
Class 0
Class 1
Class 2
Class 3
Palette
Group templates
None
Template 1
Template 2
Template 3
Create
Delete
Manual cell class
Current class: 0
Item editor
General
Name: Group 6
Region
Base: 711 px 820 px
Size: 183 px 507 px
Plane
Columns Rows
Size: 48 128
Is AND plane
Vertical
Inverted inputs
Internal offset: 0
Group
Columns Rows
Size: 12 32
X Y
Cell size: 15,27 15,86
Cell padding: 0 4 5 0
Cell orientation: Vertical
Class count: 4
Use template reference
Plot class reference
Copy reference from template
Columns Rows
Binary start: 24 64
Binary size: 12 64
Class orientation: Vertical
Start End
Discard bits: 0 0
Test output
Plane 1rows: 128 columns: 48 overlap:0 incomplete: 0

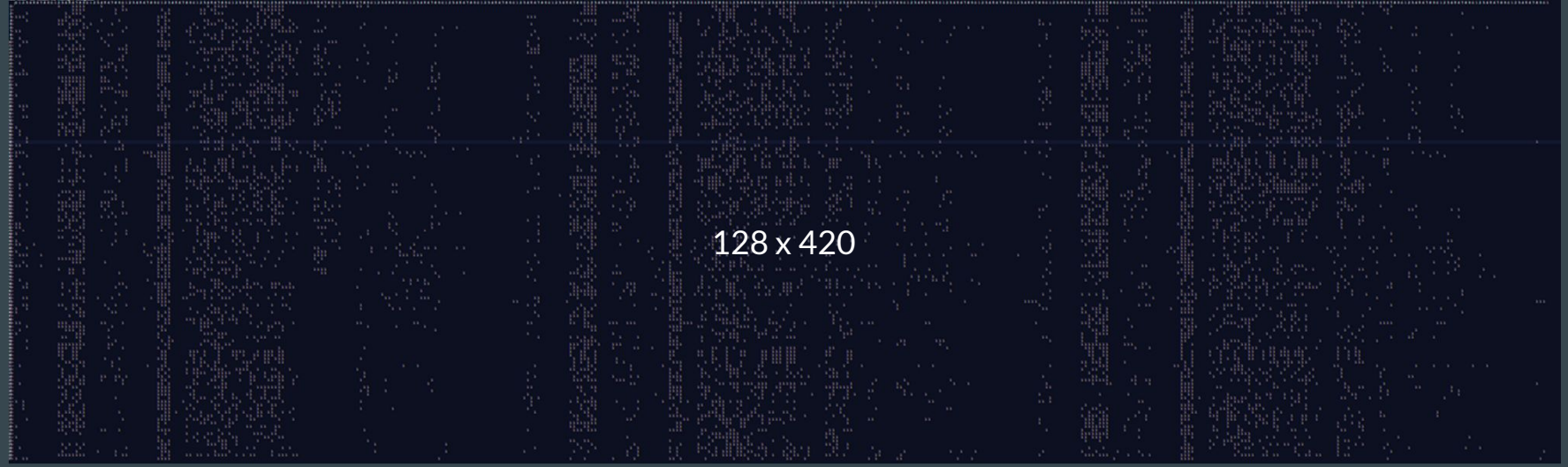
# pladecode

## https://github.com/peterbjornx/pladecode

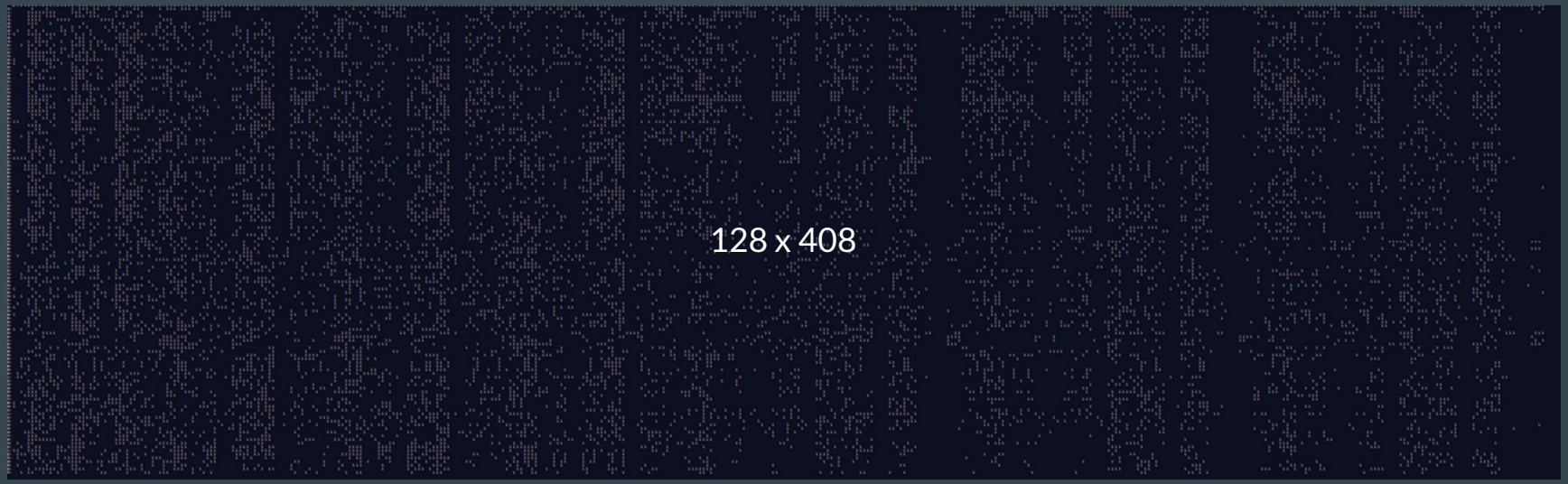
- Decodes mask-programmed ROM and PLAs
- Outputs
  - C simulator code for PLA
  - Text representation of ROM
- Qt based UI exposes all state

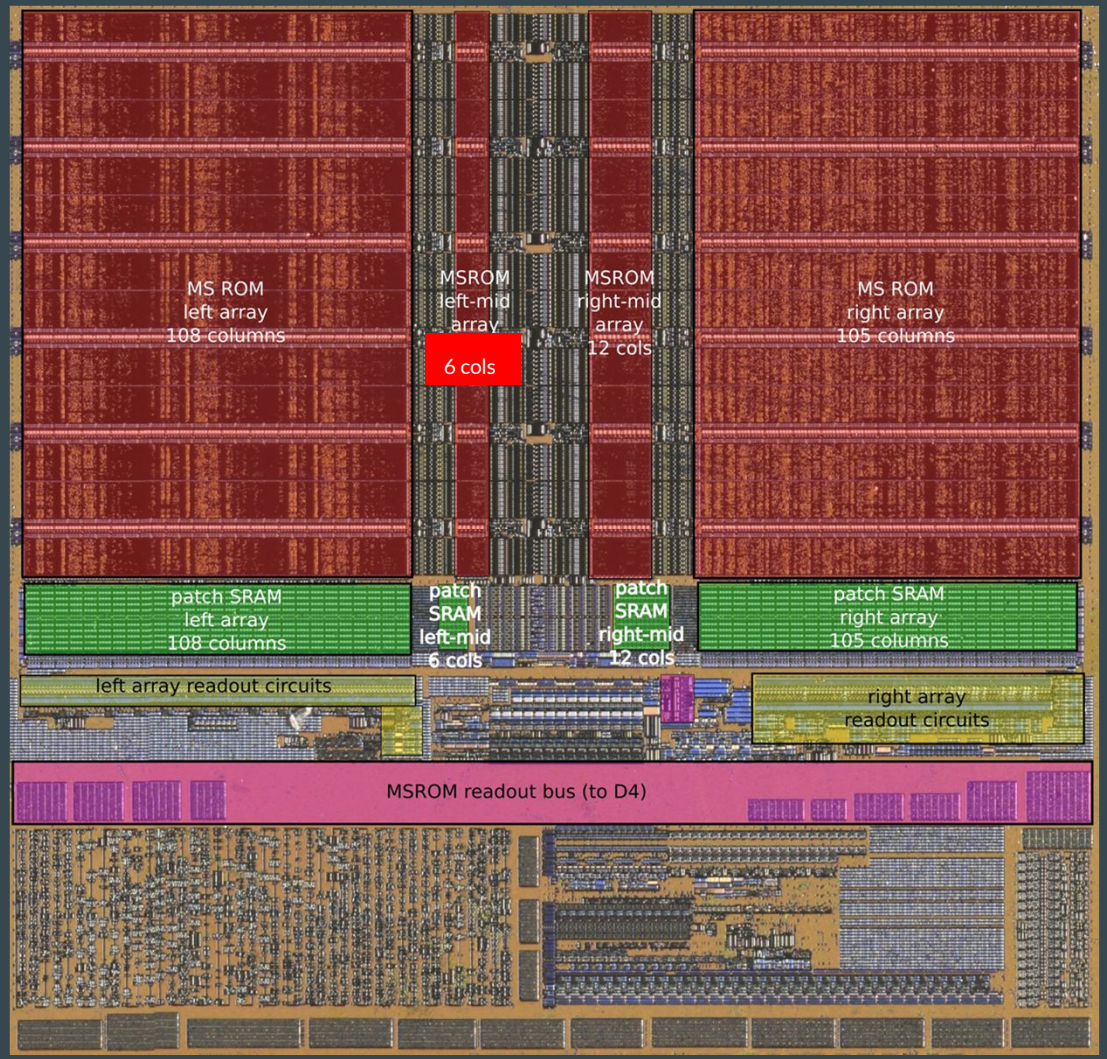


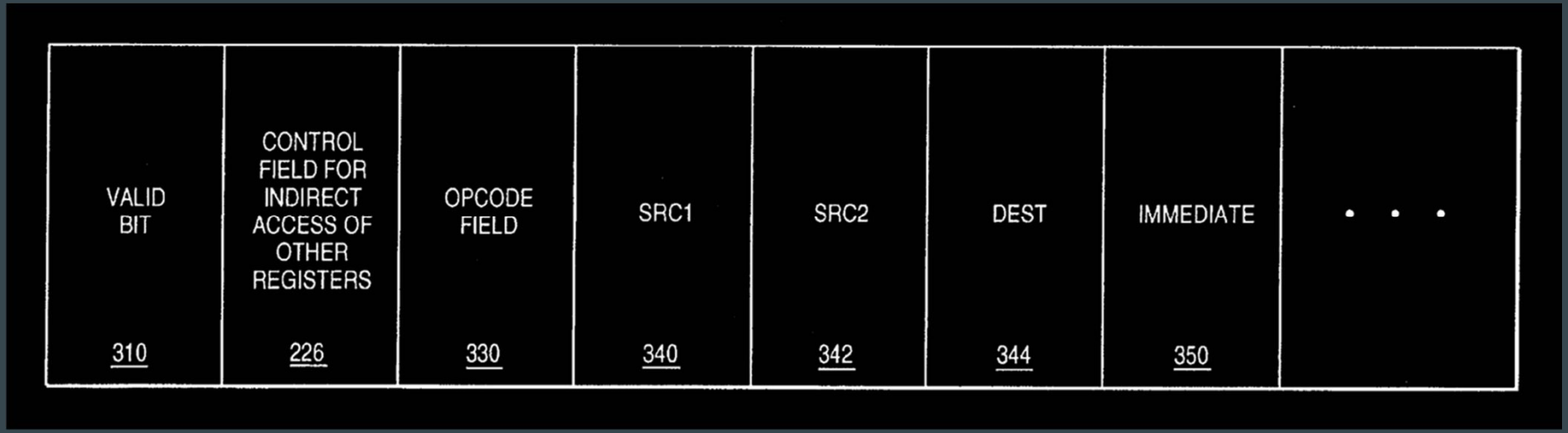
L



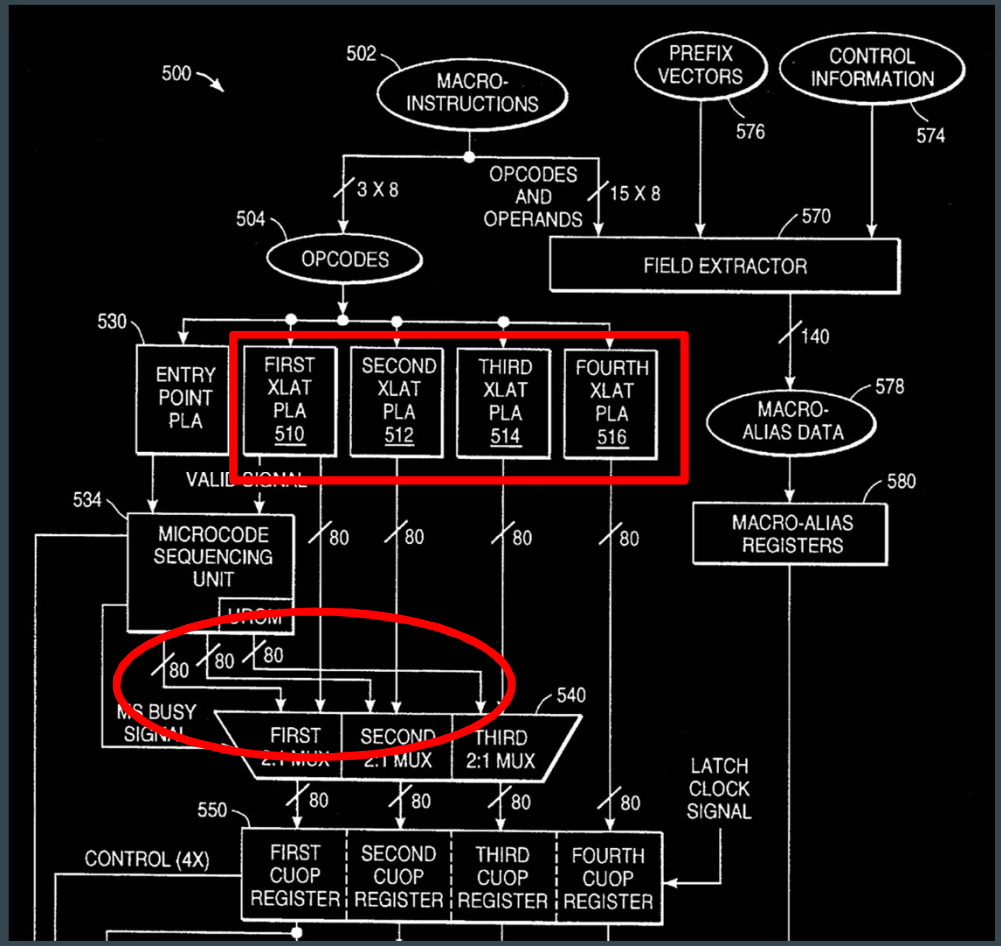

R



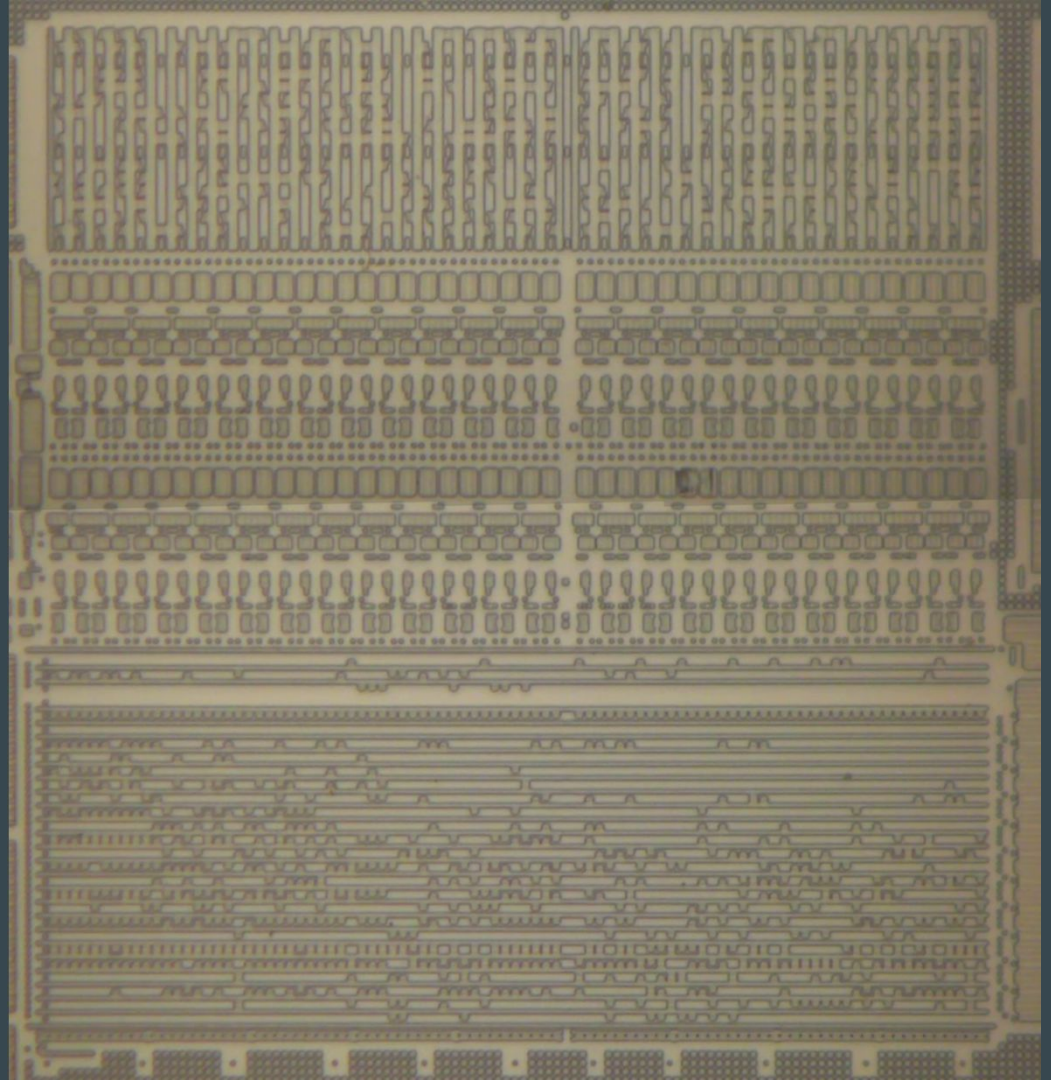

MS ROM
left array
108 columns
MSROM
left-mid
array
6 cols
MSROM
right-mid
array
12 cols
MS ROM
right array
105 columns
patch SRAM
left array
108 columns
patch
SRAM
left-mid
6 cols
patch
SRAM
right-mid
12 cols
patch SRAM
right array
105 columns
left array readout circuits
right array
readout circuits
MSROM readout bus (to D4)

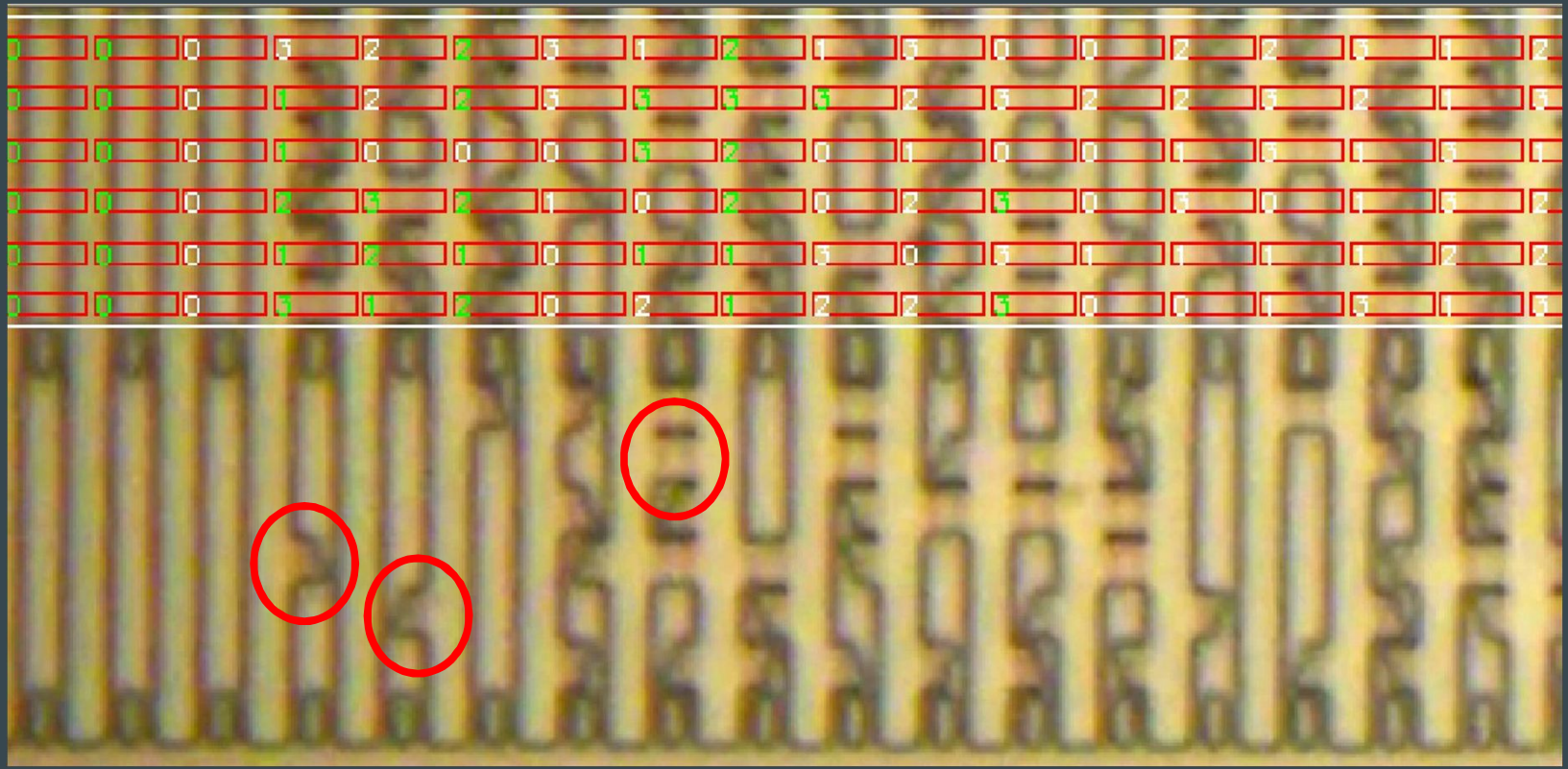


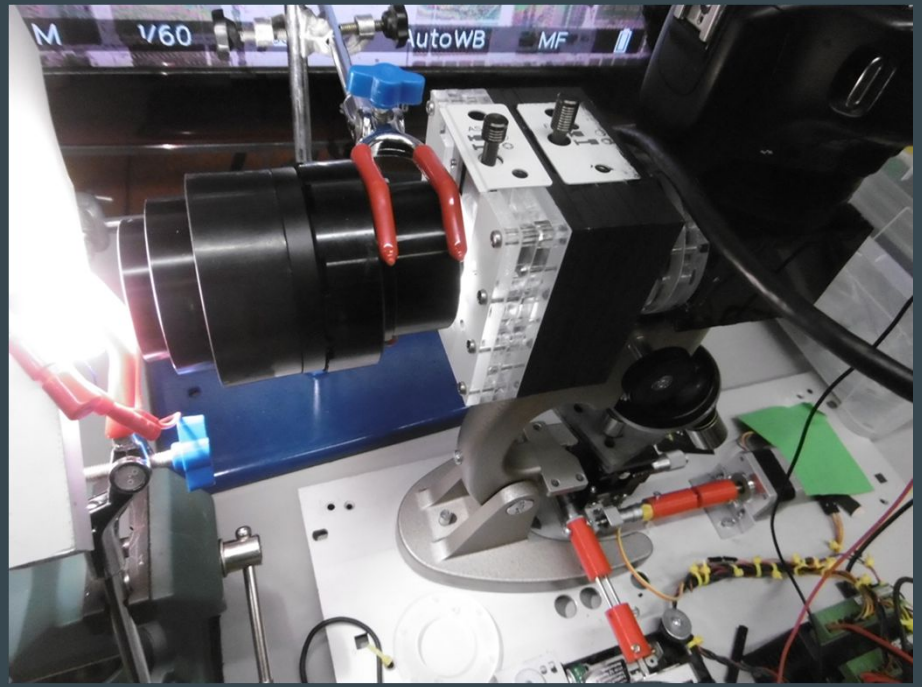

Source: US5559974, Fig 3




Source: US5559974, Fig 5

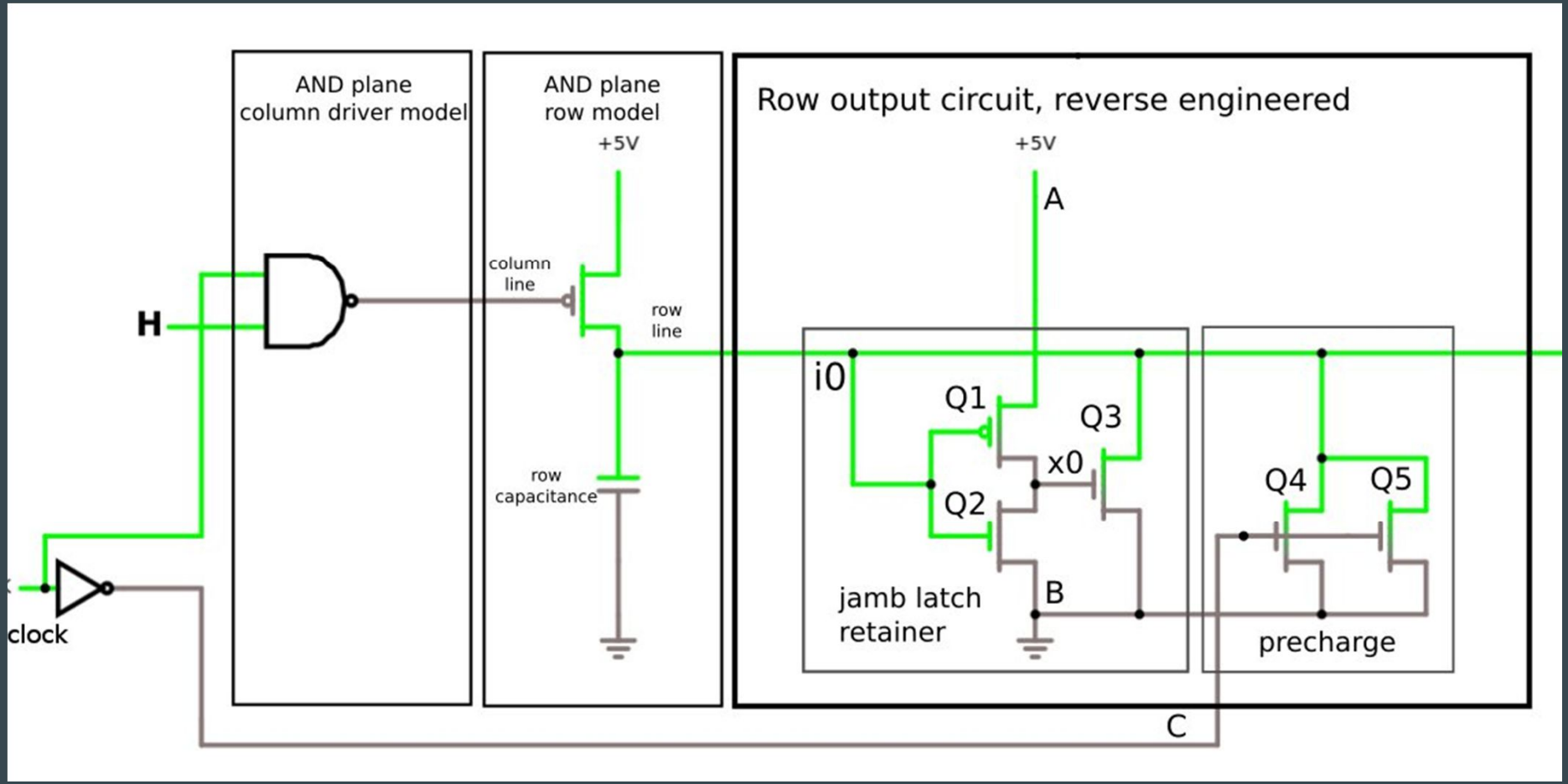


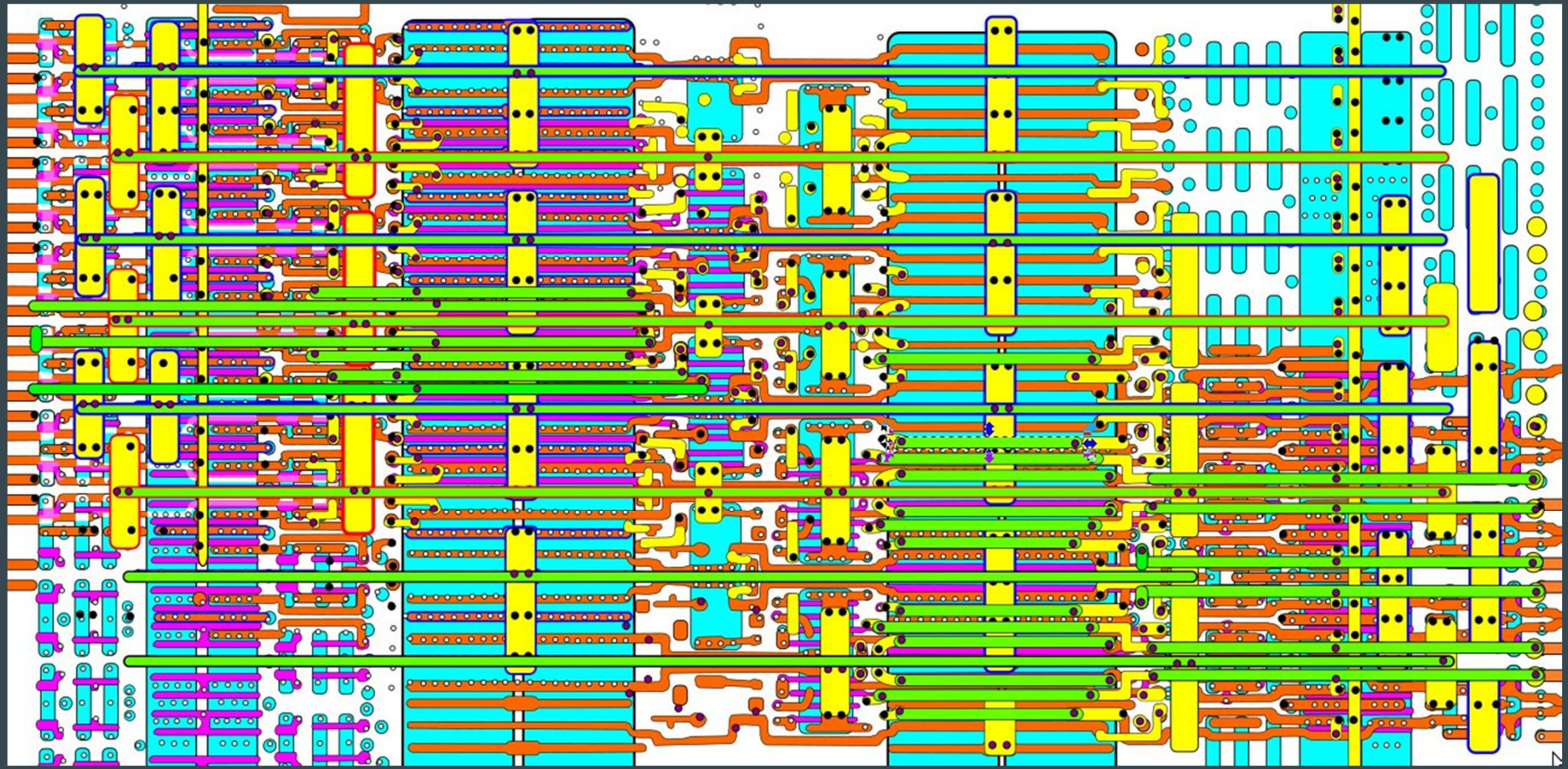


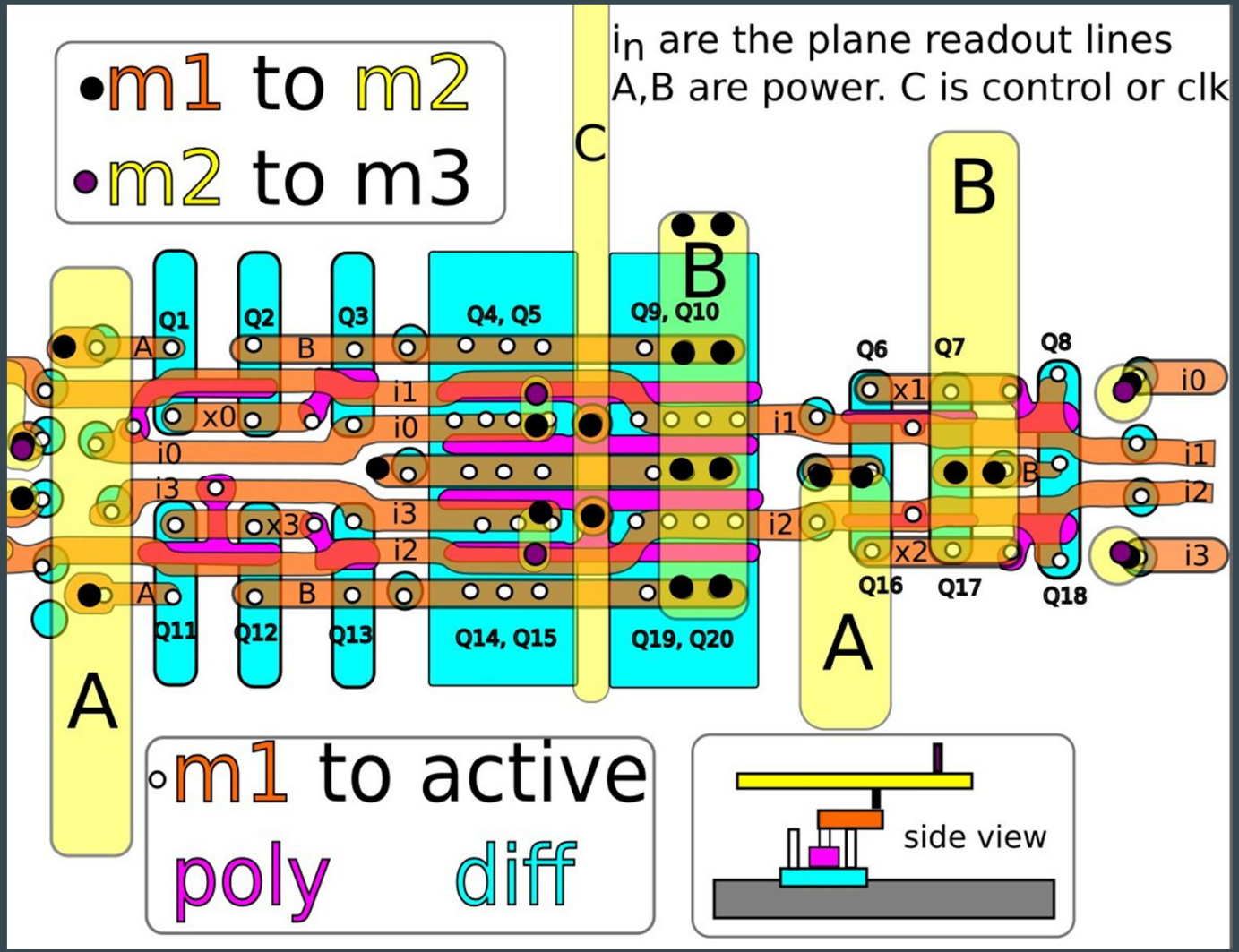

M
V60
AutoWB
MF

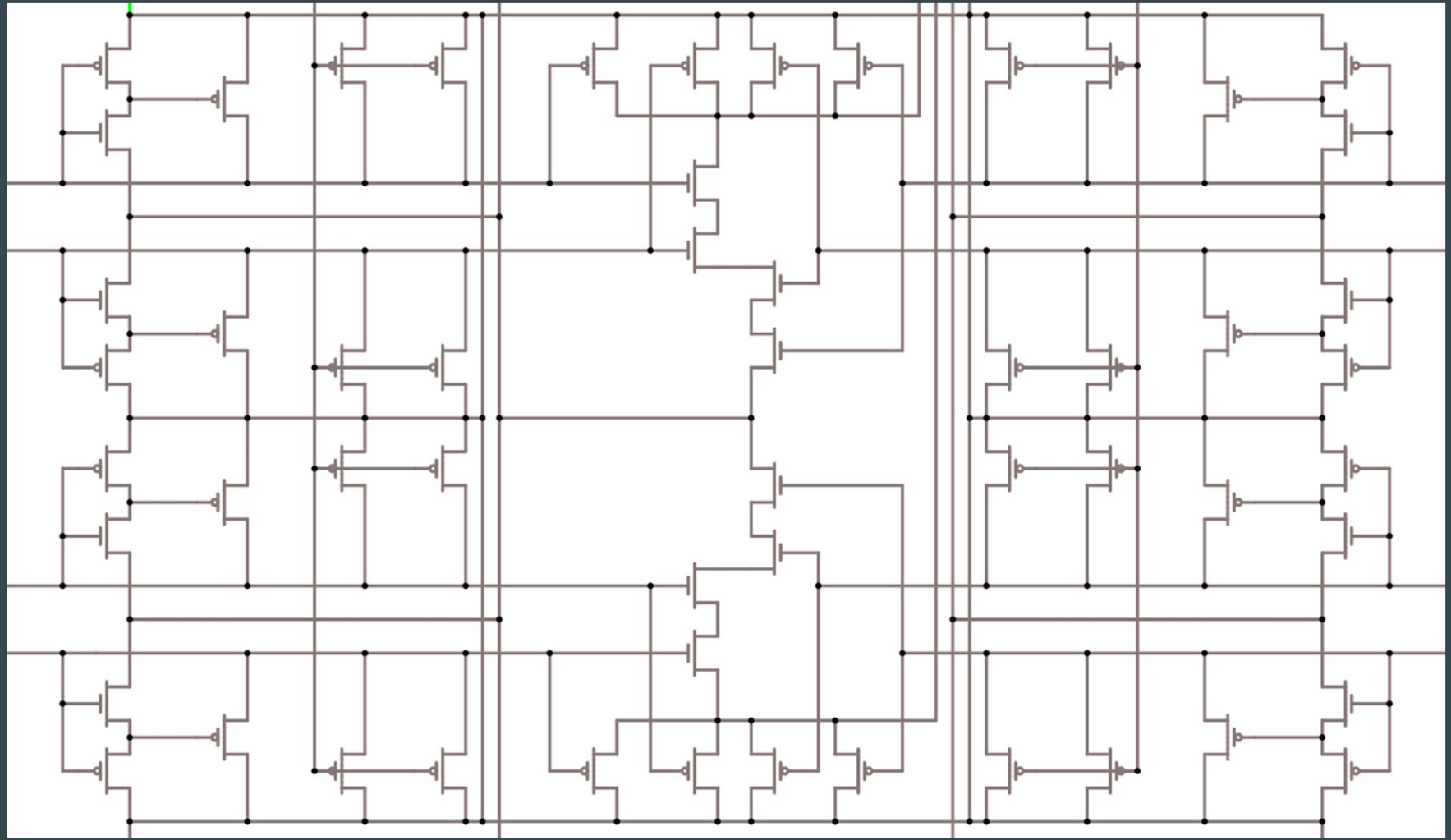

HG
koper
super glans
polish
140 ml

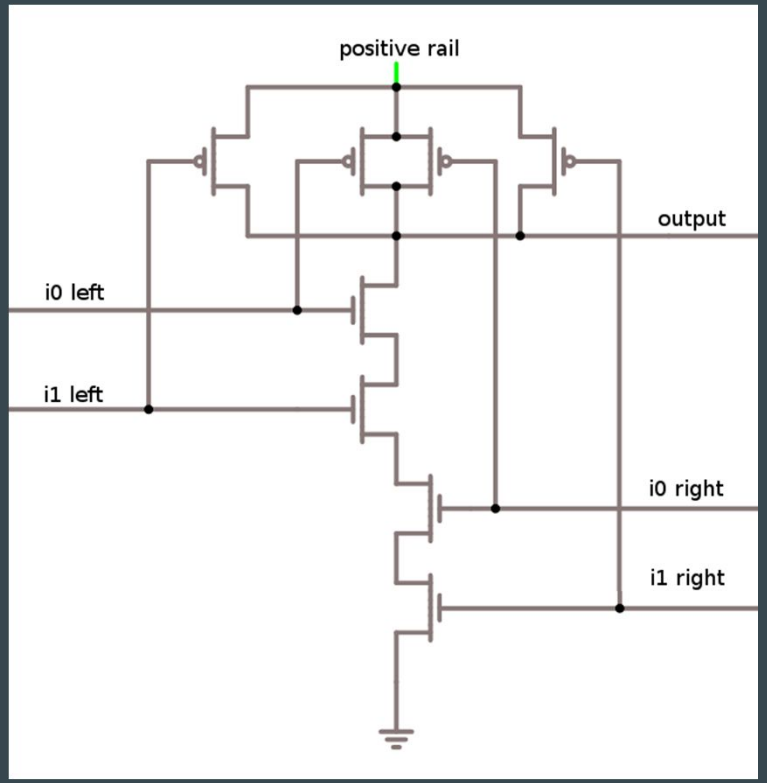

70
0

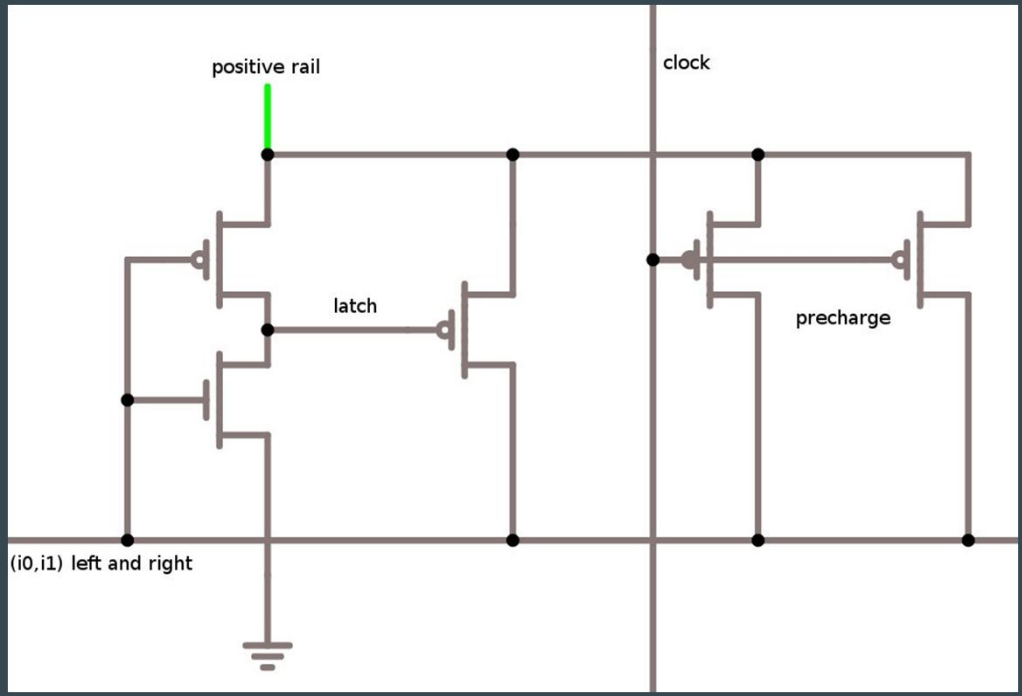

AND plane
column driver model
AND plane
row model
+5V
column
line
row
line
row
capacitance
Row output circuit, reverse engineered
+5V
A
H
i0
Q1
Q3
x0
Q2
Q4
Q5
jamb latch
retainer
B
precharge
clock
C

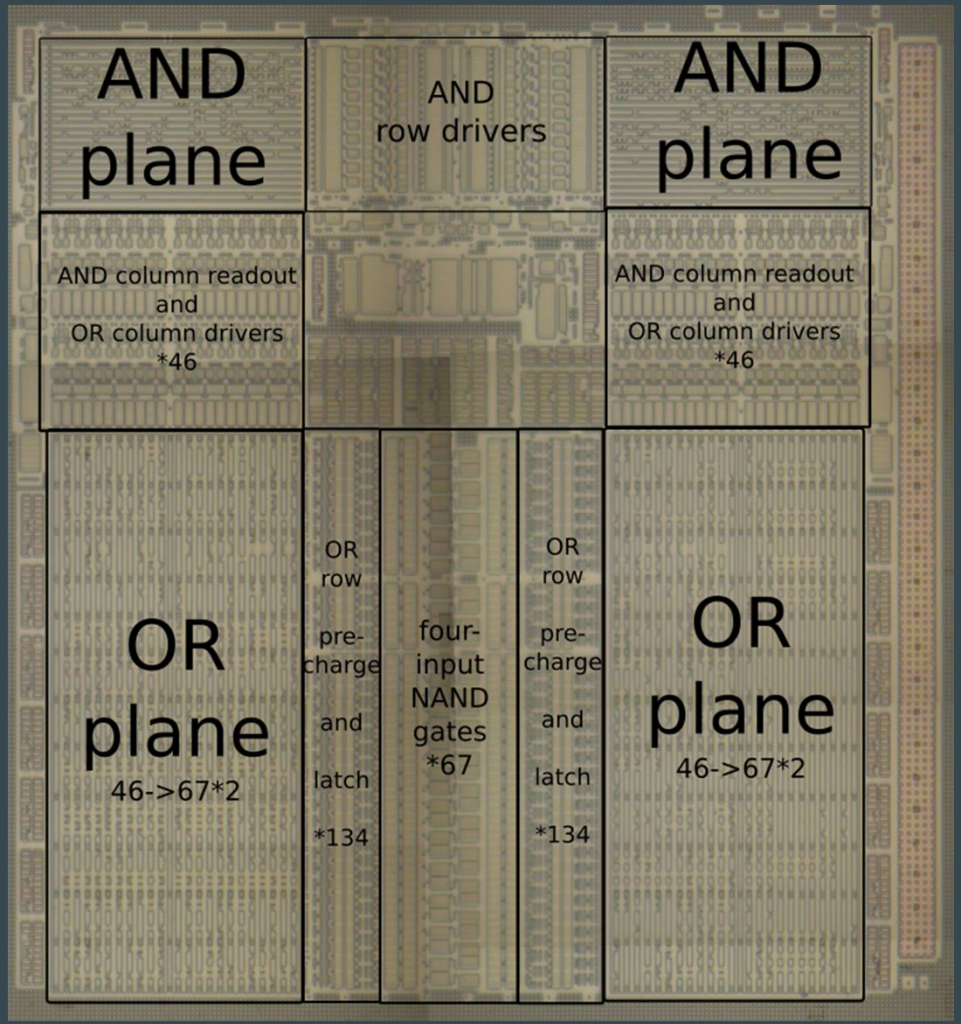



•m1 to m2
•m2 to m3
in are the plane readout lines
A,B are power. C is control or clk
C
B
B
A
A
Q1
Q2
Q3
Q4, Q5
Q9, Q10
Q6
Q7
Q8
Q11
Q12
Q13
Q14, Q15
Q19, Q20
Q16
Q17
Q18
A
B
x0
x3
x1
x2
i0
i1
i2
i3
•m1 to active
poly
diff
side view

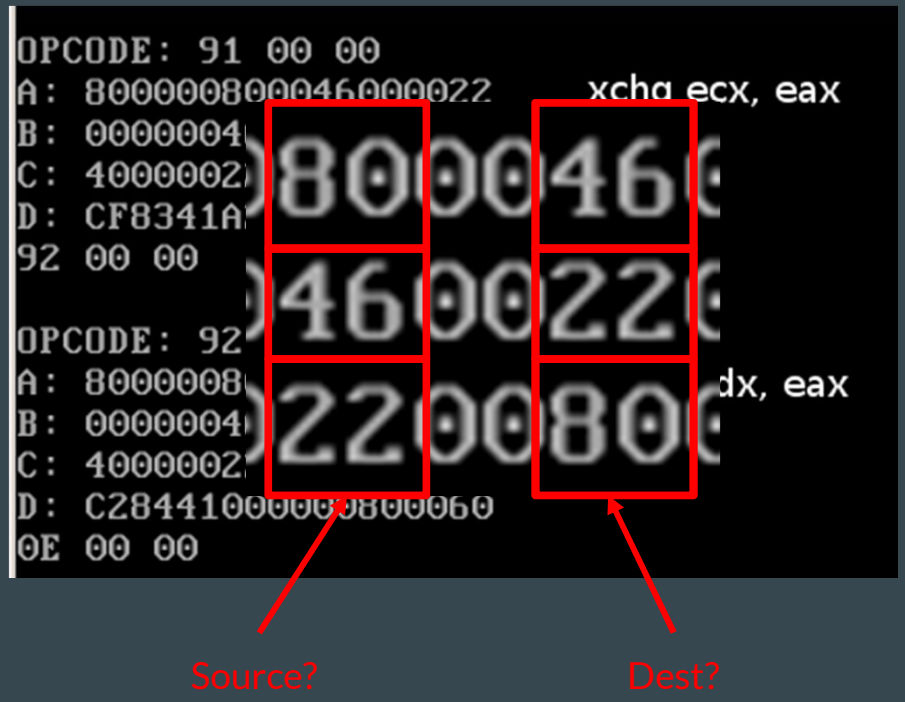



positive rail
output
i0 left
i1 left
i0 right
i1 right


positive rail
clock
latch
precharge
(i0,i1) left and right

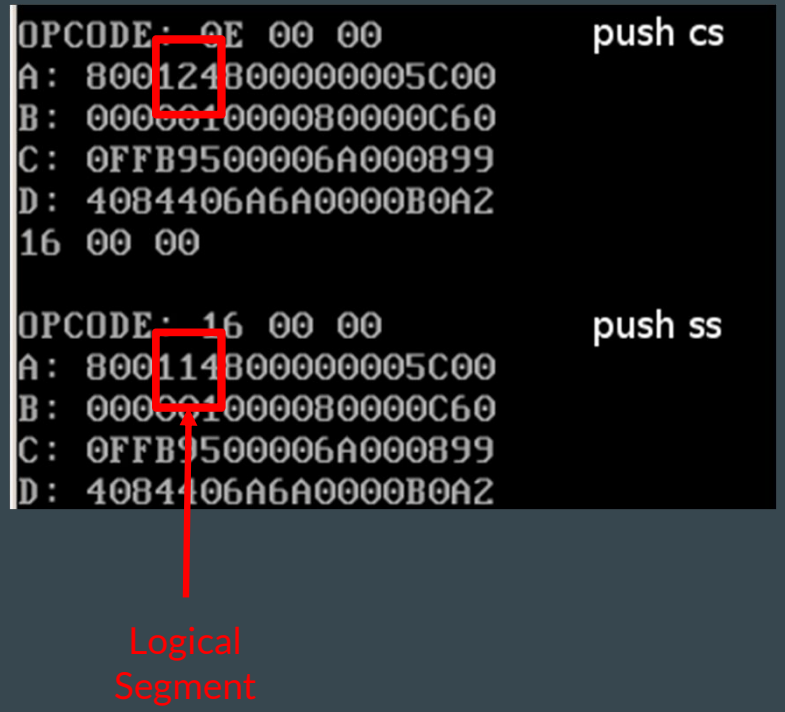

AND plane
AND row drivers
AND plane
AND column readout and OR column drivers *46
AND column readout and OR column drivers *46
OR plane
46->67*2
OR row pre-charge and latch *134
four-input NAND gates *67
OR row pre-charge and latch *134
OR plane
46->67*2

# Known semantics: XCHG




```
t = a
a = b
b = t
```

# Finding fields by comparison

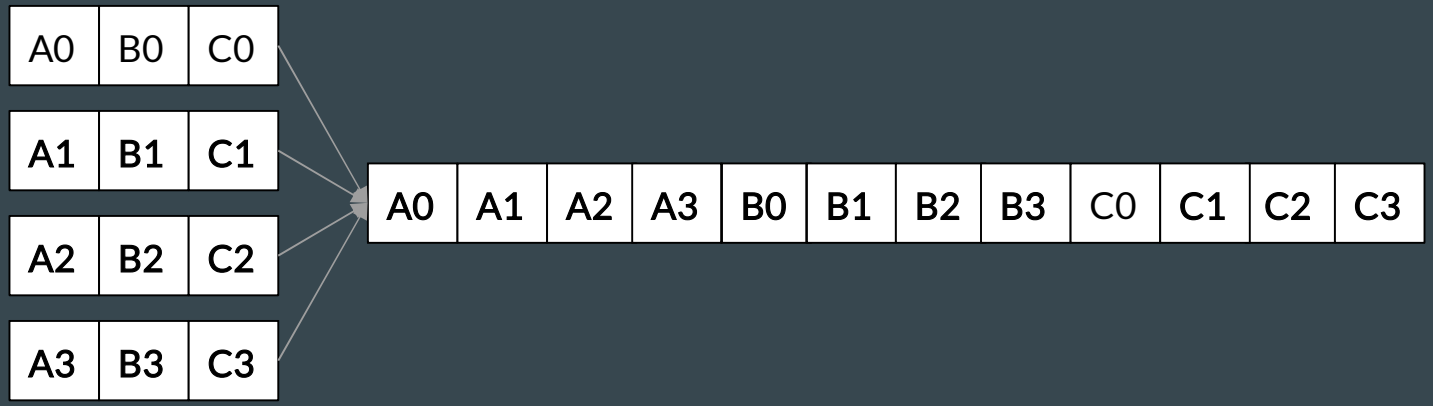


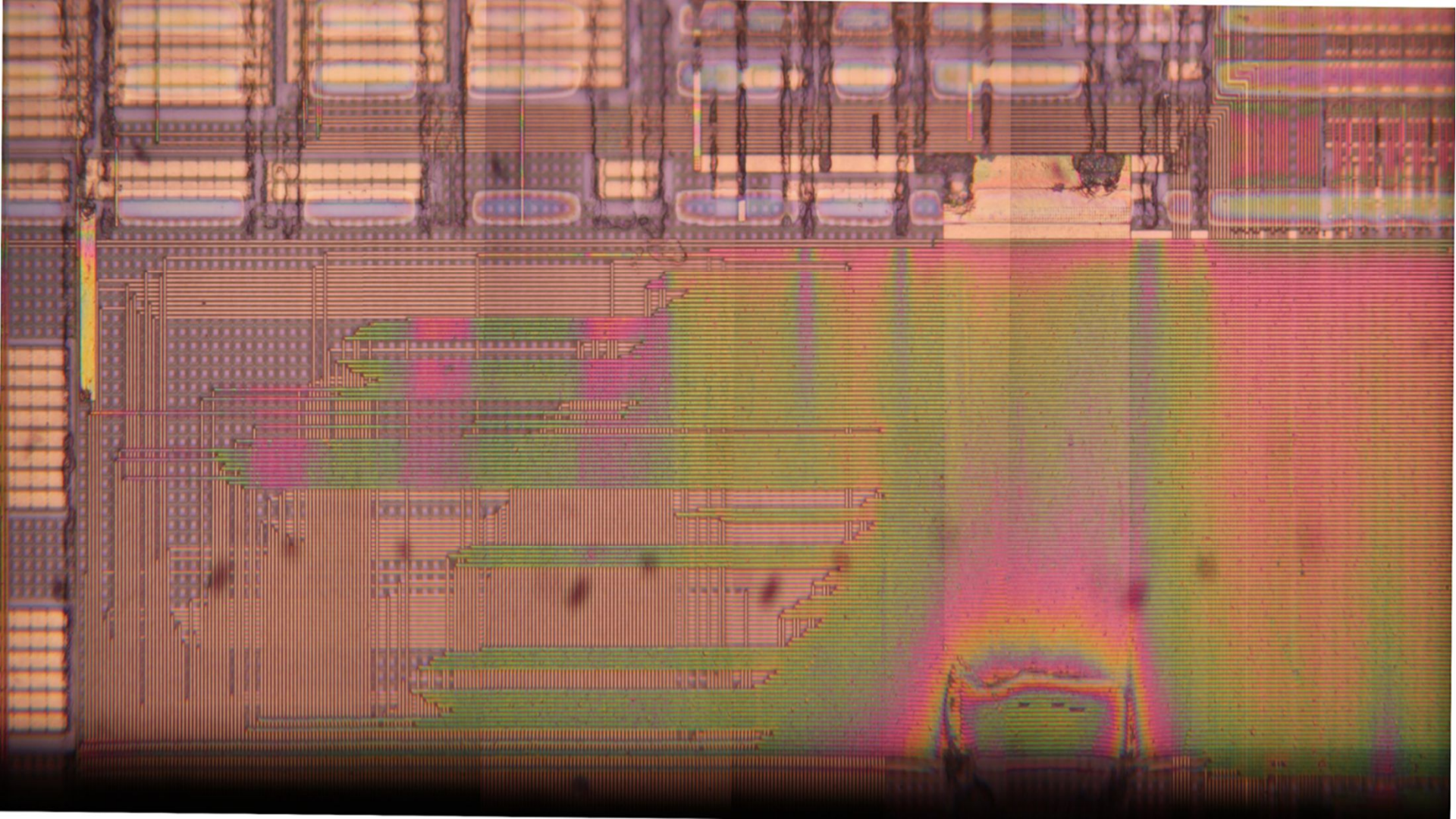

```
t = getsegsel($seg)
store...
```

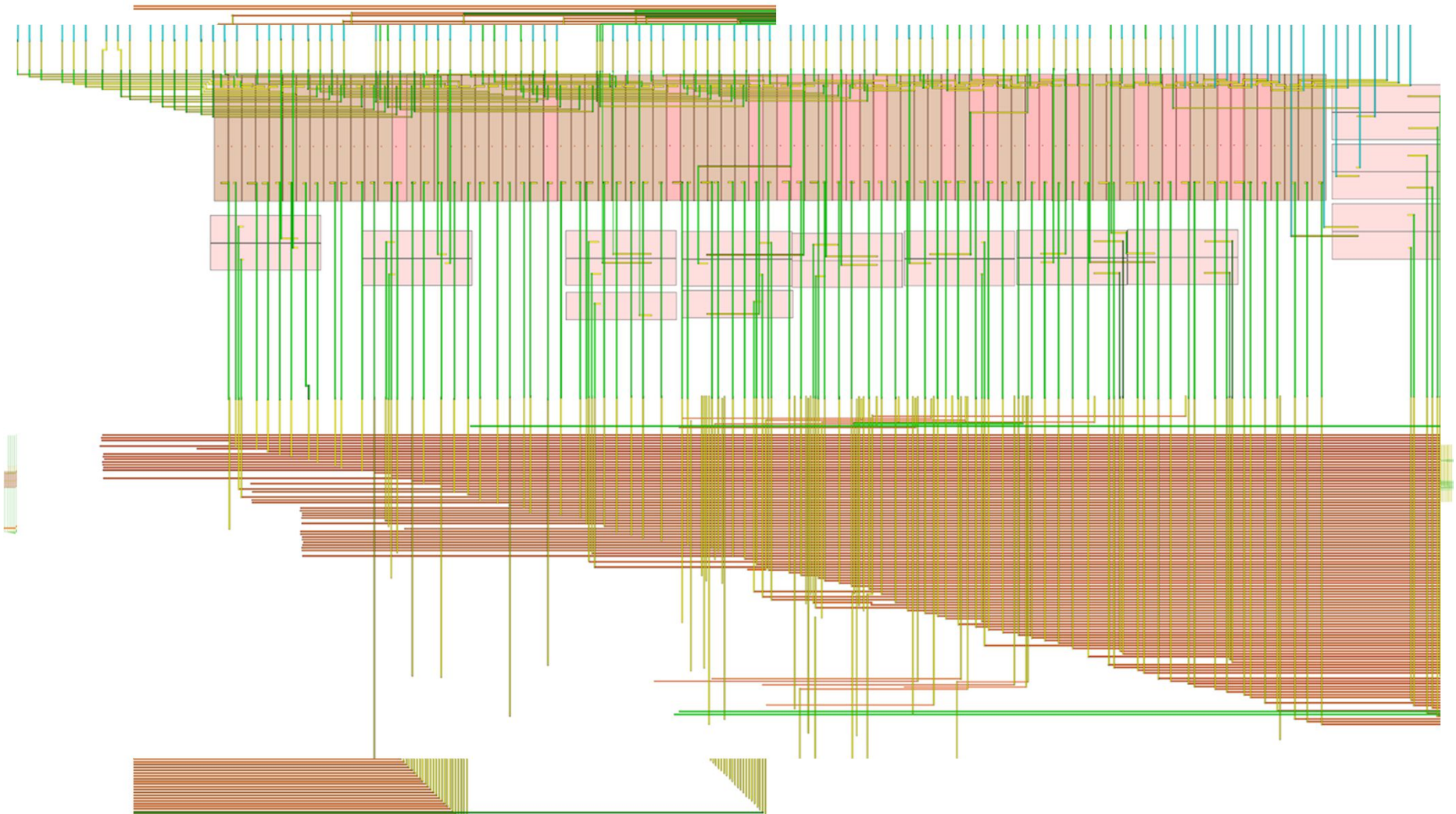
# Mapping this onto the ROM

- 432+24+48+420 =  924 columns per physical row
- 72 bit microinstruction word
- 3 microinstructions per cycle from ROM
- 128 rows per block

# Row interleaving







```
Indexing rectangles...
    found 111 named rectangles after metal 1
    found 111 named rectangles after metal 2
    found 111 named rectangles after metal 3

Assigning rectangle ids...
    (re)named 177 rectangles, for a total of 288 in metal 1
    (re)named 1531 rectangles, for a total of 1531 in metal 2
    (re)named 1357 rectangles, for a total of 1357 in metal 3

Indexing vias...
    found 0 named vias after via 1 to 2
    found 0 named vias after via 2 to 3

Assigning via ids...
    (re)named 158 vias, for a total of 2 in via 1 to 2 layer
    (re)named 252 vias, for a total of 2 in via 2 to 3 layer

Indexing nets...
    found 160 nets after metal 1
    found 216 nets after metal 2
    found 216 nets after metal 3
    found 216 nets after via 1 to 2
    found 216 nets after via 2 to 3

Assigning rectangles to nets...
    assigned 288 rectangles in metal 1, generating 70 new nets
    assigned 1531 rectangles in metal 2, generating 996 new nets
    assigned 1357 rectangles in metal 3, generating 1152 new nets

Assigning vias to nets...
    assigned 158 vias in via 1 to 2, generating 158 new nets
    assigned 252 vias in via 2 to 3, generating 252 new nets

Initial index and assign pass results:
    total 3176 rectangles across all layers
    total 410 vias across all layers
    total 2844 nets, of which 2628 newly autogenerated

Determining intra-layer connectivity...
    found 58 already connected, 11 new endpoint joins in metal 1, merging away 11 nets
    found 453 already connected, 125 new endpoint joins in metal 2, merging away 125 nets
    found 84 already connected, 108 new endpoint joins in metal 3, merging away 108 nets

Inferring vias...
    found 596 already connected, 421 new endpoint joins between metal 1 and 2, of which 126 already had vias
    found 557 already connected, 1172 new endpoint joins between metal 2 and 3, of which 172 already had vias

Determining via connectivity...
    found 0 already connected, 451 new between metal 1 and via 1 to 2, merging away 451 nets
    found 426 already connected, 27 new between metal 2 and via 1 to 2, merging away 27 nets
    found 7 already connected, 1204 new between metal 2 and via 2 to 3, merging away 1204 nets
    found 308 already connected, 885 new between metal 3 and via 2 to 3, merging away 885 nets

Flushing net information to rectangles already in SVG...
    updated 70 nets in metal 1
    updated 996 nets in metal 2
    updated 1152 nets in metal 3

Flushing net information to vias already in SVG...
    updated 2 nets in via 1 to 2
    updated 142 nets in via 2 to 3
    found 1295 vias not in SVG

Adding new vias to SVG...

Writing output file svg-edited3.svg...
pbx@theseus:/media/pbx/29CD-5993/d4p$
```

| | | | | | |
|---|---|---|---|---|---|
| 0 | x63 | 3 | y63 | 6 | z63 |
| 1 | x65 | 4 | y65 | 7 | z65 |
| 2 | x58 | 5 | y58 | 8 | z58 |
| | | | | | |
| 9 | x64 | 21 | y64 | 33 | z64 |
| 10 | x60 | 22 | y60 | 34 | z60 |
| 11 | x61 | 23 | y61 | 35 | z61 |
| 12 | x57 | 24 | y57 | 36 | z57 |
| 13 | | 25 | | 37 | |
| 14 | x67 | 26 | y67 | 38 | z67 |
| 15 | x59 | 27 | y59 | 39 | z59 |
| 16 | x56 | 28 | y56 | 40 | z56 |
| 17 | x66 | 29 | y66 | 41 | z66 |
| 18 | x70 | 30 | y70 | 42 | z70 |
| 19 | x4 | 31 | y4 | 43 | z4 |
| 20 | x6 | 32 | y6 | 44 | z6 |
| | | | | | |
| 45 | x12 | 65 | y12 | 85 | z12 |
| 46 | x5 | 66 | y5 | 86 | z5 |
| 47 | x17 | 67 | y17 | 87 | z17 |
| 48 | x11 | 68 | y11 | 88 | z11 |
| 49 | x10 | 69 | y10 | 89 | z10 |
| 50 | x18 | 70 | y18 | 90 | z18 |
| 51 | x19 | 71 | y19 | 91 | z19 |
| 52 | x9 | 72 | y9 | 92 | z9 |
| 53 | x7 | 73 | y7 | 93 | z7 |
| 54 | x20 | 74 | y20 | 94 | z20 |
| 55 | x41 | 75 | y41 | 95 | z41 |
| 56 | x8 | 76 | y8 | 96 | z8 |
| 57 | x39 | 77 | y39 | 97 | z39 |
| 58 | x42 | 78 | y42 | 98 | z42 |
| 59 | x68 | 79 | y68 | 99 | z68 |
| 60 | x40 | 80 | y40 | 100 | z40 |
| 61 | x43 | 81 | y43 | 101 | z43 |
| 62 | x71 | 82 | y71 | 102 | z71 |
| 63 | x49 | 83 | y49 | 103 | z49 |
| 64 | x21 | 84 | y21 | 104 | z21 |

# Google find!

AP-526

intel®

## D.3. Uop Pseudocode for Macroinstructions

Following is a description of the timing tables.

Example timing:

HDR: "IMUL rm32" : 1111011.1 11.101.sss ------

- FLOW tmp0 := int_mul.port0.latency4(EAX, REG_sss)

  EAX := move.port01.latency1(tmp0);

  EDX := port0.latency1(Tmp0, const);

Where:

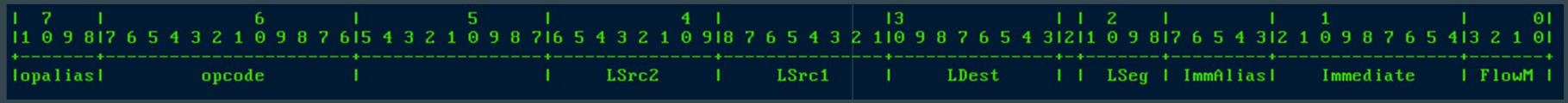
"IMUL rm32"

Is the Intel Macro Instruction:

# Putting it to use

```
HDR: "STD":                ( 11111101 -------- -------- )
1FLOW: TMP5= Port_01.latency_1(ArithFLAGS, SystemFlags)
2FLOW: TMP5= Port_01.latency_1(TMP5, 000010000) # 0x10
3FLOW: SystemFlags= Port_01.latency_1(TMP5, 000001010) #0xA
4FLOW:     sink=     move.Port_01.latency_1(CONST)
OPCODE: FD 00 00
A: 01C0088F9283800001 TMP5        := SystemFLAGS,ArithFLAGS  OP:1C0 LSEG:0 IMM:000 f0:1 f1: 0 f2:0 f3: 11 f4:0
B: 06060000038382C100 TMP5        := CONST      ,TMP5        OP:606 LSEG:0 IMM:010 f0:0 f1:16 f2:0 f3:  0 f4:0
C: 06050400038F82C0A0 SystemFLAGS := CONST      ,TMP5        OP:605 LSEG:0 IMM:00A f0:0 f1:16 f2:0 f3:  8 f4:0
D: 060010000000800002 SINK        := CONST      ,CONST       OP:600 LSEG:0 IMM:000 f0:2 f1: 0 f2:0 f3: 20 f4:0
```

# The microinstruction encoding, so far.




- The obvious 3 operand form fields: opcode, src1, src2, dest
- FlowMarker:
  - Indicates Beginning Of Macroinstruction (BOM), End of Macroinstruction (EOM) and other flow control metadata

# Instructions and Opcodes

LEA: This instruction is optionally included in the hybrid execution unit **35**. If it is to be included, the hybrid execution unit **35** will also have to include a 3 input adder which, in a preferred embodiment, would consist of a 3-to-2 reducer leading into a 2-input Kogge-Stone adder. The LEA instruction returns an effective address calculated from a base which is on the lower bits of source **2**, and index which is on source **1** and a displacement which is on the upper bits of source **2** (also referred to as LEA source **3**), and a scale which is bits **[5:4]** of the Uopcode. The result is right aligned to bit **0** of the result. The base is shifted left by the amount of the scale and added to the index and the displacement. The size of the data returned is determined by the LEA modifier. The ASZ16 modifier returns 16 bit data. The ASZ32 modifier returns 32 bit data. For each modifier, the upper data bits are forced to zero. No flags are returned. If implemented, this instruction follows the following formula:

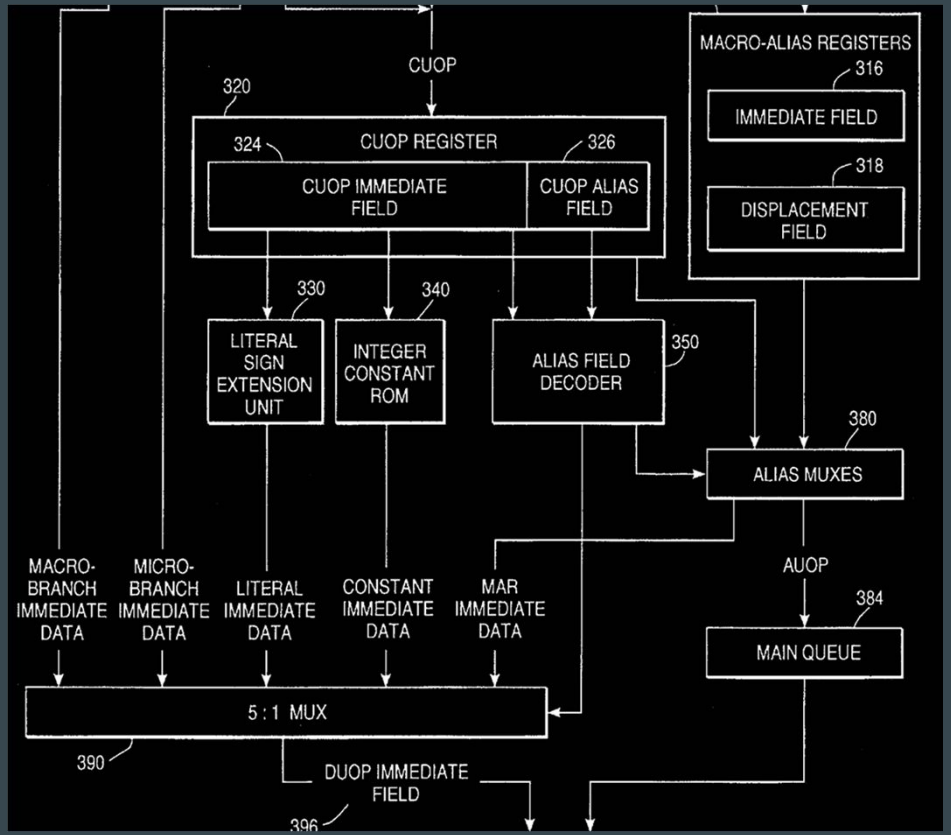
```
        flags := '0
    LEA.ASZ16
        data := '0::70 & (src1[15:0] + src2[15:0] SHL
        uop[5:4] + src2[47:32])
    LEA.ASZ32
        data := '0::54 & (src1[31:0] + src2[31:0] SHL
        uop[5:4] + src2[63:32])
```

INTEXTRACT: This instruction returns portions of the source **2** input data right aligned to bit **[0]** of the result, according to the INTEXTRACT modifier. The HI32 modifier returns source 2 bits **[63:32]**. The HI16 modifier returns source **2** bits **[31:16]**. The UP32 modifier returns source **2** bits **[66:35]**. For each modifier, the upper data bits are forced to zero. The flags associated with the source **2** input data are returned. The result data is set as follows:

```
        flags := src2 flags
    INTEXTRACT.HI32
        data out := '0::54 & src2[63:32]
    INTEXTRACT.HI16
        data out := '0::70 & src2[63:48]
    INTEXTRACT.UP32
        data out := '0::54 & src2[66:35]
```

**US5574942A**

# Immediate Operands



# Immediate Operands

- Only 9 bits for Immediate
- Immediate Alias Control field selects source for “immediate” data
  - 0x11 seems to select Macroinstruction Alias, IMM field selects which MAR register
    - 0x0 Macroinstruction Immediate
    - 0x10 REG_Op_Size (Operand size in bytes)
    - 0x11 virt_ip
    - 0x12 next_virt_ip
- 0x04 seems to be used for sign-extended literal, where the data is signext(IMM)
- 0x16, 0x0E also seem to be literal?
- Constant ROM?

# Registers: LSrc1,2 and Dest

| Index | 0x00 | 0x08 | 0x10 | 0x18 | 0x20 | 0x28 | 0x30 | 0x38 |
|---|---|---|---|---|---|---|---|---|
| 0 | CONST | AL | ST(0) | | | AX | | EAX |
| 1 | SINK | CL | ST(1) | | | CX | | ECX |
| 2 | TMP0 | DL | ST(2) | | | DX | | EDX |
| 3 | TMP1 | BL | ST(3) | | | BX | | EBX |
| 4 | TMP2 | AH | ST(4) | | FCC | SP | | ESP |
| 5 | TMP3 | CH | ST(5) | | ArirthFlags | BP | | EBP |
| 6 | TMP4 | DH | ST(6) | FSW | | SI | | ESI |
| 7 | TMP5 | BH | ST(7) | SystemFlags | | DI | | EDI |

# Registers: LSrc1,2 and Dest

| Index | 0x50 | 0x70 | 0x88 | 0xA8 | 0xC0 | 0xC8 |
|---|---|---|---|---|---|---|
| 0 | | | (E)AX | (E)AX | | REG_sss |
| 1 | | | (E)CX | (E)CX | | |
| 2 | | MMX Source | (E)DX | (E)DX | | |
| 3 | | | (E)BX | (E)BX | | |
| 4 | ST(i) | | (E)SP | (E)SP | Reg in Opcode | REG_ddd |
| 5 | | | (E)BP | (E)BP | | |
| 6 | | MMX Dest | (E)SI | (E)SI | | |
| 7 | | | (E)DI | (E)DI | | |

# Segments

| Index | 0x00 | 0x08 |
|---|---|---|
| 0 | SEG_SINK | ES |
| 1 | | CS |
| 2 | | SS |
| 3 | | DS |
| 4 | | FS |
| 5 | | GS |
| 6 | GDTR | |
| 7 | LDTR | TR |

| |
|---|
| IDTR? |
| LINSEG? |
| PHYSEG? |

# Control registers

Shown microcode is CNL, from Github

```
U24ad   wrmsr_apicbase_continue:
U24ad           BTS_DSZ64                               TMP5        < SRC_SXQA      , TMP5          , TMP5          , K0            #
U24ae           UJMPCC_DIRECT_TAKEN_CONDNB                          < CONST_0       , TMP5          , UROM_SINK     , K0            # , XR: U884c wrmsr_past_rsvd_bit_check
U24af           MOVEFROMCREG_DSZ32                      TMP9        < CONST_0       , 0x6f7         , TMP9          , K0            # Lit in SRC_SXQA,  0x6f7 = CORE_CR_RR_MODE
U24b0           ANDOR_DSZ32                             TMP9        < TMP2          , TMP9          , TMP9          , K0            #
U24b1           MOVEFROMCREG_DSZ32                      TMPD        < CONST_0       , 0x2ec         , TMPD          , K0            # Lit in SRC_SXQA,  0x2ec = PMH_CR_SMRR_BASE
U24b2           MOVEFROMCREG_DSZ32                      TMP7        < CONST_0       , 0x2ee         , TMP7          , K0            # Lit in SRC_SXQA,  0x2ee = PMH_CR_SMRR_MASK
U24b3           UCALLPARAM_DIRECT                                   < CONST_0       , CONST_0       , UROM_SINK     , K0            # , XR: Ua050 check_apic_base_overlap_with_smrr
U24b3           IMPLIED JUMP TO Ua050 check_apic_base_overlap_with_smrr
--------------------------------------------------------------------------------

U24b4           MOVEFROMCREG_DSZ32                      TMPD        < CONST_0       , 0x2ff         , TMPD          , K0            # Lit in SRC_SXQA,  0x2ff = PMH_CR_SMRR2_BASE
U24b5           MOVEFROMCREG_DSZ32                      TMP7        < CONST_0       , 0x2f2         , TMP7          , K0            # Lit in SRC_SXQA,  0x2f2 = PMH_CR_SMRR2_MASK
U24b6           UCALLPARAM_DIRECT                                   < CONST_0       , CONST_0       , UROM_SINK     , K0            # , XR: Ua050 check_apic_base_overlap_with_smrr
U24b6           IMPLIED JUMP TO Ua050 check_apic_base_overlap_with_smrr
--------------------------------------------------------------------------------
```

# Some example microcode flows: DIV

```
TMP2 := WUCONCAT( EDX, EAX )
TMP0 := DIV( TMP2, REG_sss )
EAX  := MOVE( TMP2 )
EDX  := INTEXTRACT.HI32( TMP0 )
```

# Some example microcode flows

| 7FA4h | SS base |
|---|---|
| 7FA8h | ES |
| 7FACh | CS |
| 7FB0h | SS |
| 7FB4h | DS |
| 7FB8h | FS |
| 7FBCh | GS |
| 7FC0h | LDTR |
| 7FC4h | TR |
| 7FC8h | DR7 |
| 7FCCh | DR6 |
| 7FD0h | EAX |
| 7FD4h | ECX |
| 7FD8h | EDX |
| 7FDCh | EBX |
| 7FE0h | ESP |
| 7FE4h | EBP |
| 7FE8h | ESI |
| 7FECh | EDI |
| 7FF0h | EIP |
| 7FF4h | EFLAGS |
| 7FF8h | CR3 |
| 7FFCh | CR0 |

```
TMP7        := OP_032       (CONST_0e_171   , CONST_0e_171   ) IMM: 171
TMP2        := ADD_DSZM     (TMP7           , CONST_0c_07c   ) IMM: 7c U2: 20
TMPB        := LOAD_200     (CONST_06_000   , TMP2           ) LSeg: SEG_01
REG_36      := LOAD_200     (0x-0000004     , TMP2           ) IMM: 1fc LSeg: SEG_01
SINK        := MOVETOCREG   (CONST_0e_06b   , REG_36         ) IMM: 6b
TMP4        := LOAD_200     (0x-0000008     , TMP2           ) IMM: 1f8 LSeg: SEG_01
TMP3        := LOAD_200     (0x-000000C     , TMP2           ) IMM: 1f4 LSeg: SEG_01
EDI_30      := LOAD_200     (0x-0000010     , TMP2           ) IMM: 1f0 LSeg: SEG_01
ESI_30      := LOAD_200     (0x-0000014     , TMP2           ) IMM: 1ec LSeg: SEG_01
EBP_30      := LOAD_200     (0x-0000018     , TMP2           ) IMM: 1e8 LSeg: SEG_01
ESP_30      := LOAD_200     (0x-000001C     , TMP2           ) IMM: 1e4 LSeg: SEG_01
EBX_30      := LOAD_200     (0x-0000020     , TMP2           ) IMM: 1e0 LSeg: SEG_01
EDX_30      := LOAD_200     (0x-0000024     , TMP2           ) IMM: 1dc LSeg: SEG_01
ECX_30      := LOAD_200     (0x-0000028     , TMP2           ) IMM: 1d8 LSeg: SEG_01
EAX_30      := LOAD_200     (0x-000002C     , TMP2           ) IMM: 1d4 LSeg: SEG_01
REG_36      := LOAD_200     (0x-0000030     , TMP2           ) IMM: 1d0 LSeg: SEG_01
SINK        := MOVETOCREG   (CONST_0e_17d   , REG_36         ) IMM: 17d
TMP9        := LOAD_200     (0x-0000034     , TMP2           ) IMM: 1cc LSeg: SEG_01
REG_36      := LOAD_200     (0x-0000038     , TMP2           ) IMM: 1c8 LSeg: SEG_01
TMPA        := WRSEGFLD     (0x00000004     , REG_36         ) IMM: 04 LSeg: TR
REG_36      := LOAD_200     (0x-000003C     , TMP2           ) IMM: 1c4 LSeg: SEG_01
TMPA        := WRSEGFLD     (TMPA           , REG_36         ) IMM: 04 LSeg: LDTR
REG_36      := LOAD_200     (0x-0000040     , TMP2           ) IMM: 1c0 LSeg: SEG_01
TMPA        := WRSEGFLD     (TMPA           , REG_36         ) IMM: 04 LSeg: GS
REG_36      := LOAD_200     (0x-0000044     , TMP2           ) IMM: 1bc LSeg: SEG_01
TMPA        := WRSEGFLD     (TMPA           , REG_36         ) IMM: 04 LSeg: FS
REG_36      := LOAD_200     (0x-0000048     , TMP2           ) IMM: 1b8 LSeg: SEG_01
TMPA        := WRSEGFLD     (TMPA           , REG_36         ) IMM: 04 LSeg: DS
REG_36      := LOAD_200     (0x-000004C     , TMP2           ) IMM: 1b4 LSeg: SEG_01
TMPA        := WRSEGFLD     (TMPA           , REG_36         ) IMM: 04 LSeg: SS
TMP1        := LOAD_200     (0x-0000050     , TMP2           ) IMM: 1b0 LSeg: SEG_01
REG_36      := LOAD_080     (0x-00000DC     , TMP2           ) IMM: 124 LSeg: SEG_01
TMPA        := MOVETOCREG   (CONST_0e_101   , REG_36         ) IMM: 101
REG_36      := LOAD_200     (0x-0000054     , TMP2           ) IMM: 1ac LSeg: SEG_01
TMPA        := WRSEGFLD     (TMPA           , REG_36         ) IMM: 04 LSeg: ES
```

# Some example microcode flows

```
TMPA    := OP_032        (CONST_0e_022   , CONST_0e_022   ) IMM: 22
TMPA    := BTEST_DSZN    (TMPA           , 0x00D          ) IMM: 0d
SINK    := OP_313        (TMPA           , TMP6           ) IMM: 86
TMP3    := RDSEGFLD      (0x00000001     , 0x00000001     ) IMM: 01 LSeg: TR
TMP3    := BTEST_DSZN    (TMP3           , 0x00B          ) IMM: 0b
SINK    := OP_313        (TMP3           , TMP8           ) IMM: 199 U2: 8
TMP3    := LOAD_100      (0x00000066     , 0x00000066     ) IMM: 66 LSeg: TR U2: 20
TMP4    := SHR           (TMP1           , 0x003          ) IMM: 03
TMP5    := AND_DSZN      (TMP1           , 0x007          ) IMM: 07
TMP2    := SUB_DSZN      (CONST_0e_004   , TMP2           ) IMM: 04
TMP6    := SHR           (CONST_0e_00f   , TMP2           ) IMM: 0f
TMP6    := OP_624        (TMP6           , TMP5           )
TMP5    := LOAD_100      (TMP4           , TMP3           ) LSeg: TR
TMP6    := AND_DSZN      (TMP5           , TMP6           )
SINK    := OP_315        (TMP6           , TMP8           ) IMM: 199
SINK    := OP_098        (CONST_0        , REG_33         )
SINK    := SETcc_NS      (CONST_0        , REG_33         )
TMP5    := MOVE_DSZN     (CONST_0        , ESI_30         )
TMP3    := MOVE_DSZN     (CONST_0        , EDI_30         )
TMP7    := MOVE_DSZ16    (CONST_0        , ECX_30         )
SINK    := MOVETOCREG    (CONST_0e_16e   , REG_33         ) IMM: 16e
SINK    := OP_290        (CONST_0        , TMP0           )
TMP3    := MOVE_DSZN     (0x0C1          , 0x0C1          ) IMM: c1
TMP4    := OP_120        (0x00D          , 0x00D          ) IMM: 0d
SINK    := SIGEVENT      (TMP4           , TMP3           )
```

# Some example microcode flows

```
EBX_30      := MOVE_DSZN     (CONST_0       , CONST_0       )
ECX_30      := MOVE_DSZN     (CONST_0       , CONST_0       )
EDI_30      := MOVE_DSZN     (CONST_0       , CONST_0       )
ESI_30      := MOVE_DSZN     (CONST_0       , CONST_0       )
ESP_30      := MOVE_DSZN     (CONST_0       , CONST_0       )
EBP_30      := MOVE_DSZN     (CONST_0       , CONST_0       )
EDX_30      := OP_032        (CONST_0e_18c  , CONST_0e_18c  ) IMM: 18c
REG_34      := OP_52F        (0x1F0         , 0x1F0         ) IMM: 1f0
EDX_30      := OR_DSZN       (EDX_30        , 0x060         ) IMM: 60
DH          := MOVE_DSZ8     (0x006         , 0x006         ) IMM: 06
REG_37      := MOVE_DSZN     (0x0FF         , 0x0FF         ) IMM: ff U2: 20
```

# Future work

- Complete ROM extraction:
  - Finish capturing all ROM blocks (3/6 done)
  - Determine uopcode[6] column
- Map out more opcodes and registers
- Determine Entry Point PLA input format and extract macro-op entry points
- Find constant ROM
- Map out CRBUS addresses
- Determine update encryption mechanism

# Acknowledgements


- Martijn Boer (https://www.flickr.com/people/sic66/)
  - Provided me with the high-resolution scans of the ROM and PLAs
- RevSpace (Hackerspace The Hague)
  - for being a community full of great ideas and a place to work on such complicated projects.
- Alyssa Milburn (@noopwafel)
  - who keeps taunting me with crazy project ideas.
- Peter Cywinski
  - whose auction site trawling got me the microscope upgrades that made this possible
- Many others